Elements of a C++ program

Basic elements

- `#include` directives
  - declares functions, objects and classes that are being used
  - ex. `iostrem` allows use of `cout`, provides further support of console input / output
- `using` statement
  - enables all names within a given namespace to be referenced directly
  - not strictly necessary buy a major convenience
  - without the `using` statement we would have to refer to `cout` with `std::cout`, the qualified name
- `main` function
  - the part of the program that actually does something

General Structure of a C++ program

1. Declarations, `include`.
2. `using` statement.
3. Type declarations, including classes.
4. Global variable decorations.
5. Function prototypes.
6. Function definitions including `main`.

Namespaces

Namespaces can be used more explicitly by referring to objects in the namespace directly. For example instead of `using namespace std;`, `using std::cout;` can be used to refer to the `cout` object.

The number of libraries available in C / C++ is quite large, which meant when many libraries are in use within a program, there is the potential for name conflicts.

The `using` statement grants access to all the symbols in a library only within the scope in which it is defined.

Namespaces can be defined with the `namespace` keyword

Data

Two basic types: primitive (built-in) types and user-defined types. There is no difference between the two except that primitive types are pre-defined.

Mixing Numeric Types

C++ issues a warning when a data type is converted to a type with a smaller range. Ex: Float -> Int

Signed numbers are expressed as the bitwise negation + 1. For example -15 is expressed as:

- `0000 0000 0000 1111` (15)
- `1111 1111 1111 0000` (bitwise negation)
- `1111 1111 1111 0001` (+1) # Operators

Data form the simplest expressions, the use of operators creates larger expressions. Terminating assignment with a semicolon turns it into a statement. To create complex expressions, one must understand:

- Precedence: which operators have priority over others. Ex: `*` has precedence over `+`.
- Associativity: what happens when there are two or more operators at the same level of precedence. Ex: `5 - 5 - 5` is the same as `(5 - 5) - 5` an not `5 - (5 - 5)`.

Modifier Precedence:

1. Scope operator
2. Data access modifiers
3. Prefix operators
4. Pointer to member
5. Multiplication and Division
6. Addition and Subtraction
7. Shift operators
8. Less than / Greater than.
9. Test for Equality
10. Bitwise and Logical conjunctions
11. Conditional operator
12. Assignment operator
13. `throw` operator
14. Join operator (`,`)

Details

- The scope operator `::` has the highest precedence
- Postfix operators `lavalue++`, `lavalue--` first pass on the current value then increment / decrement
  - Postfix operator is slightly inefficient as it creates a copy of the thing being incremented.
- Prefix modifiers associate R -> L ex: `++lvalue` increments first, `&lvlaue`, the address operator takes the address of an item from memory

- Assignment operator associates R -> L, ex: `++(bigger = big = 100)` results in `bigger = 101` and `big = 100`.
- `throw` raises an exception
- join operator combines multiple operations, ex: `while(cin >> n, n > 0)`

Cast operators

- `static_cast`
  - produce value of an expression in the new data format.
  - commonly used to suppress warnings and provide clarification
- `reinterpret_cast`
  - casts from one pointer (address) expression to another
  - i.e. leaves the data at the address given unchanged and interprets that data as the given cast type.
  - used to cast `void*` pointer to more specific type and for reading or writing binary data.
- `const_cast`
  - adds or removes `const` or `volatile` attributes
- `dynamic_cast`
- C-language cast
  - syntax for old C-language cast.
  - easier to use than other casts, but casts should be self-documenting.

Control Structures

Determines what the program does next. C++ statement syntax is recursive, each item can be used within another statement.

Summary

- `;` Null statement, performs no action
- `{ ... }` Compound statement (block)
- `data declaration`
- `if (condition) statement`
- `if` (condition) statement else statement‘
- `while (condition) statement`
- `do statement while (condition);`
- `for (int; condition; increment) statement`
- `for (type variable : container) statement`
- `switch (value) { statements }`
- `case value: statement`
- `default: statement`
- `label: statement`
- `goto label;`

- `continue;`
- `break;`
- `return;`
- `return expression;`

Exception handling

Provides a mechanism for dealing with runtime errors and other special events. Most exceptions are runtime errors, arithmetic / overflow. Exceptions handle these exceptions at runtime, respoinding to them immdeiatley.

All exceptions are derived from the base class `std::exception`. The `what()` method can used to retrieve a C-string describing the excption.

Logic-Error Exceptions

- `std::logic_error`: erros in library / operator functions not caught by the complier
- `std::runtime_error`: common runtime errors
- `std::bad_cast`: reports invalid use of `dynamic_cast` expression
- `std::bad_typeid`: reports the use of the `type_id` operator on an object that has a `void` type
- `std::domain_error`: voilation of a precondition assumed bya function
- `std::invalid_argument`: invalid argument, normally caught by the complier
- `std::length_error`: attempt to create an object larger than the physical size supported
- `std::out_of_range`: attempt tot use an arguemtn outside the allowable range

Runtime-Error Exceptions

- `std::bad_alloc`: faiure to allocate requested memroy
- `std::overflow_error` arithmetic overflow of a floating point number
- `std::range_error` reports results that fall outside the allowable range
- `std::underflow_error` floating point numbers that are too tiny to be stored in the supported range

Syntax

```
try { statements; }
[ catch (excption_class object) { statements;} ] ...
[ finally { statements } ]
```

Example

```
#include <stdexcept>
#include <iostream>

using namespace std;
```

```
...

try {
    // some stuff
} catch (exception e) {
    cout << e.what() << endl;
}
```

Functions

There are three steps in using names functions:

1. Prototype the function
2. Define the function
3. Call the function

A function prototype gives type information for the function but does not define it. A function definition also provides type information but define what the function does

Examples of prototypes

```
int a_func(int n);
int b_func(int n);
```

We can also define prototypes without specifying the argument names, i.e. `int a_func(int)`.

Typically included in a header file using

```
#include "my_proj.h"
```

Defining a function

All function definitions are also declarations, they also include statements that define what the function does

```
return_type name(args) {
    statements
}
```

- Unless a function has a `void` return type, it should always return some data.
- Arguments are passed by value, meaning functions get their own copy of the data. Arrays are the exception
- Variables declared in functions are local
  - They have automatic storage, meaning memory is allocated when the function is called, and removed when the function terminates.

  - The `static` keyword can be used to persist the variable through the life of the program

Complete function syntax

`[modifiers] return_type name(args) [override] [const] [final]`

Modifiers

- `const`: cannot modify any of the class data. In effect the function is read-only.
- `constexpr`: compiler treats return values from the functions as compile-time constants
- `explicit`: when applied, the constructor function is not used to supply an implicit conversion function.
- `final`: applied to virtual functions, prevents the function from being overridden by derived classes.
- `inline`: suggestion to complier to make the function inline, when a function is inlined the body of the function is expanded into the body of the calling function, removing function-call overhead.
- `override`: member functions only, specifies that the function overrides a function specified in the base class.
- `static`: when applied to a global function is visible with the current source file only. When applied to a member function, the function becomes static, so that it cannot access other class members except those declared as static.
- `virtual`: address not resolved until runtime

Function overloading

- Function can have the same name as long as they can be differentiated by their argument lists.

In some cases a function can be matched to more than one call, for example

```
void func(int n);
void func(double x);
```

The compiler will make the best match it can. These functions can have different return types. Return types are not used to match the function.

Variable-Length Argument Lists

`return_type name(args ...)`

Functions can be defined to take any number of arguments. Include

```
#include <cstdarg>
```

in sources files to use these kinds of functions. Argument processing functions include:

- `va_list`: declares a list name
- `va_start`: initialises the argument list, up to some last named argument
- `va_arg`: returns the next argument from the variable argument list
- `va_end`: terminates reading the argument list

Lambda functions

```
[closure] (args) { statements }
```

`[]` intended literally, `[closure]` is an optional list of variables.

Return types are usually implicit but can be specified with

```
[closure] (args) -> return_type { statements }
```

Closures allow for variables defined in a block to be captured by the lambda expression and used internally without explicitly being passed as an argument.

Possible closure values include:

- `[]`: capture nothing
- `[=]`: capture all local variables declared in the surrounding code, but only by value
- `[&]`: all variables in surrounding scope captured by reference
- `[=, args]`: all variables captured by value, except for any args.
- `[&, args]`: all variables captured by reference, except for any args.

Mutable keyword

```
[closure] (args) mutable -> return_type { statements }
```

The `mutable` keyword is used to indicate that variables can be changed, but only as temporary variables within the lambda. Changes has no effect outside the lambda.

`constexpr` Functions

`constexpr` keyword causes the compiler to treat calls to the function as runtime constants if possible. Calls are treated as constants if the arguments supplied are also constants. These values are computed at compile time (i.e. before the program is ever run).

Storing and Returning Lambdas

When a lambda is stored variables ate captured at the point the lambda is defined. If a variable is captured by reference then a fixed value is not cap-

tured. Functions can be written that return a lambda which is done using the `<functional>` header. Ex:

```
function<return_type(arg_type_list)> function_name(args)
```

Pointers, Arrays and References

References

Variable that works as an alias for another variable. Used in function arguments to make permanent changes to a variable passed to it

Function gains direct access to the variable being passed.

```
cont type &arg
```

Arrays

Arrays are a numbered collection of elements

Range based `for`

```
int arr[] = {1, 2, 3, 4, 5};
for (int &x : arr) {
    cout << x << endl;
}
```

Range based for loop raises an error for arrays passed as arguments to functions.

Pointers

Pointers store the address of a variable

```
int i = 0;
int *p = &i;
```

The `*` or the "at" operator is used to access the address of a pointer.

```
*p = 10;
cout << i; // 10
```

Close relationship between pointers and arrays. Much more efficient to use pointer to access and modify array elements. Particularly important for arrays containing millions of element, or large multidimensional arrays.

`const` values can only be assigned to `const` pointers.

The pointer itself can also be declared as `const` e.g.

```
int * const ptr = &i;
```

Classes

Creating a class creates a new data type. Provide a mechanism to group data and functions.

Definition

Classes are defined using the `class` keyword

```
class Point {
    public:
    double x, y;
};
```

Data are referred to as "members", i.e. `x, y` are members of the `Point` class. Members are accessed with the `.`, or `->` if using a pointer to the object.

Member access

- `public`, members are accessible by users
- `private`, default when no keyword is used, members not accessible.
- `proctected`, members accessible in subclasses.

Each object have their own copy of the data members (except for static members). All functions are shared.

Functions

Can be defined outside the class using

```
void Point::set_x(double nx( {
    ...
}
```

Static Members

Data member shared by all the objects of the same class. Define once and only once outside the class.

Constructors

Initialisation function called after an object is allocated in memory. Constructors have the same name as the class itself and do not have a return type.

`class_name(args)`

The default constructor is a constructor defined without any arguments. If a constructor is not supplied, then the complier supplies a constructor that does nothing. If a constructor is provided, default or not, the compiler will not supply a constructor.

Conversion functions are implicitly defined when a object can be created implicitly. The `explict` keyword can be used to prevent the constructor from being used as a conversion function.

Another constructor is the copy constructor, which takes only an object of the same class as an argument

```
Point(const Point &p)
```

Notice the `&`, i.e. a reference is passed

Destructor

Releases resources upon object termination.

```
~Point() {
    npoints--;
    cout << "Object deleted << endl;
}
```

`this` pointer

Gives access to the members of an object

```
this->x // gives access to the x member
*this // returns object
```

Operator Overloading

Operations can be defined such as addition, subtraction, multiplication etc. on the object.

```
return_type operator op() // unary
return_type operator op(right_arg) // binary
```

Commutativity has to be implemented separately, i.e. `int + Point != Point + int`.

`friend` keyword

Operators can also be implemented outside the class i.e. as a normal function. These cases apply when mixing types i.e. defining the `*` operator for `Point` and `int`.

Global functions defined on their own do not have access to private members of the class. The `friend` operator can be used to get around this.

`=` operator

If not supplied, the complier will supply its own version. The `copy` constructor is invoked by default.

`()` operator

Used to define a function. Can be specified to define a function object. The benefit over a function is that function object have an internal state; allowing for customisation of the function.

`[]` operator

Subscript operator used to access element in an array.

```
type& operator [] (const int index)
```

A reference is returned. A `const` reference can also be returned, but this has to be defined separately. It is good practise to define both types of references, `const` and non-`const`.

`++` / `--` operators

```
class_name& operator ++ ()  // ++cls
class_name operator ++ (int)    // cls++
```

`int` is a dummy variable to distinguish the prefix and postfix versions.

Conversion functions

Data conversion between instances of the class and another type.

```
operator type ()
```

Deriving Classes

Class inheritance allows for the creation of class hierarchies.

```
class class_name : public base_class {
    ...
};
```

Inherits all members except for constructors. Multiple inheritance is also supported.

Inheritance keyword

- `public`: all members are inherited (exc. constructors). Member access is preserved
- `private`: members become private in the subclass regardless of access level.
- `protected`: private and protected members inherited as is, public become protected.

Derived classes can inherit all the constructor via

```
using base_class::base_class
```

Note (C++11)

Up-casting

A pointer of type parent can be associated with an address of the child type. The pointer can perform any functionality of the parent class

```
class Animal { ... };
class Rabbit : public Animal { ... };

Animal *ptrAnimal;
ptrAnimal = &objRabbit;
```

Virtual Functions and Overriding

Declarations may declare members that are already members of the base class. Such members are said to override the old declarations. `virtual` keyword used to ensure declaration for class is called at runtime.

- Any function that might be overridden should be declared virtual
- Types must match
- `virtual` keyword only need once (for base class)

The `override` keyword can be used to self-document when a function is overridden. (C++11)

```
funciton_declaration override;
```

Bit Fields

Bit fields are class member. They are useful when space is at a premium / for manipulating individual bits.

```
unsigned field_name : integer;
```

Unions

A data type that in which the members share the same address in memory. # Preprocessor Directives

A directive is a command carried out by the preprocessor. A preprocessor decides which lines to compile, it also carries out a search-and-replace before the source is complied.

`#include` directive is the most essential - it brings in a set of declarations from another file.

General Syntax

Typically directives should be placed on their own line and begin with a `#` sign.

Summary

- `#define symbol` creates a symbolic name and assigns an empty string to it useful in conjunction with the `#ifdef` directive
- `#define symbol value` creates a symbolic name and assigns replacement text to it.
- `#define symbol(args) value` creates a macro function
- `#elif condition` starts a conditional compilation block, should follow another `#elif`, `#if` or `#ifdef`
- `#endif` ends a conditional compilation block
- `#error message` ends compilation immediately and prints the message
- `#if condition` begins a conditional compilation block
- `#ifdef symbol` begins conditional block, checks to see if specified symbol is defined
- `#ifndef symbol` begins conditional block, checks to see if specified symbol is not defined
- `#include <filename>` read contents of *file_name*
- `#include "filename"` read contents of project own header files
- `#line line_number` sets next line of code to *line_number* used for reporting purposes
- `#line line_number file_name` sets current *file_name* and *line_number* to the values indicated
- `#pragma command_text` responds to compiler-vendor-specific directive
- `#undef symbol`removes a symbol declaration

Using Directives to Solve Specific Problems

Creating Symbols

`#define` keyword useful create symbols used throughout a program

Creating Macros with `#define`

Macros are a convenience but have some drawbacks - noting they carry out search-and-replace during preprocessing - using templates or functions might be more efficient

```
#define MAX(A, B) (A > B ? A : B)
```

Conditional Compilation

Conditional compilation relies on `#if` and its variations. For example changes for specific version can be isolated and switched on/off.

```
#define VERSION 3.0

#if VERSION > 2.5
// ...
```

```
#endif
```

Preprocessor Operators

- `#macro_arg` strigifies an argument
- `token1##token2` concatenates two tokens
- `defined(args)` evaluates to true/false if the symbol has been defined

Predefined Macros

C++ preprocessor provides some predefined macros. Mainly used for printing diagnostic info during runtime.

- `assert(statement)` terminates program if statement is false, part of `<cassert>`
- `static_assert(statement, fail_msg)` similar to assert by requires no header file. Reports an error at compile time.
- `NDEBUG` turns off debugging behaviour, disabling use of `assert` macros
- `__DATE__` produces 11 char date string
- `__FILE__` produces string containing the file name
- `__LINE__` produces string containing the current lint number
- `__STDC__` defined if complier supports standard C only
- `__TIME__` produces 8 char time
- `__cplusplus` produces symbol if the C++ language is supported by the compiler

Creating Project Header Files

Header files are intended to be included by every individual source. `#define "my_proj.h"`. Should include

- prototype of each and every function intended to be shared by all modules
- `extern` declaration for all global data
- Class declarations for the project
- `enum` and `typedef` declarations
- `#define` directives

Header files should not include executable code only declarations.

```
// funciton prototypes
void do_stuff();
...

// external variables, not actually created
extern double time_left;
...
```

Creating and Using Templates

Templates provide generalised classes and functions that can be reused with different base types. Best sources of templates is the standard template library (STL).

Templates: Syntax and Overview

```
template<template_params> declaration
```

declaration is a function or class (structs and unions) work as well.

```
template<typename T>
class Vec2D {
    public:
    T x, y;
};
```

Each occurrence of `T` is a type supplied later. The value of `T` can be supplied when instantiating the object.

Function Templates

Easy to declare and use, to instantiate them they just need to be called.

```
template<typename type_args> return_type name (args) {
    ...
}
```

Dealing with Type Ambiguities

In some cases the argument types are ambiguous. A way around this is to use `static_cast` or to explicitly specify the declare the function's template parameter.

```
name<type>(args);
```

Function Templates with multiple parameters

```
template<typename T1, typename T2>
```

Class Templates

Class template builds a generalised class that can build on top of any data type. Template classes are more complex than instantiating a function template.

```
template<typename T> class class_name {
    // ...
}
```

Class Templates with Member Functions

Class templates become more useful with member function. Functions can be defined:

- inline
- as a prototype within the class and then defined outside

Syntax

```
template<typeargs>
return_type class_name<typeargs>::member_name () {
    // ...
}
```

A similar syntax applies to constructors, ex.

```
template<typename T>
Vec2D<T>::Vec2D() {
    // ...
}
```

Using Integer Template Parameters

Parameterised types are declared with the `typename` syntax, however, a template can also take integers as parameters. A common use is setting a size or limit of some kind.

Template Specialisation

In some cases it may be more beneficial to handle specific types differently - known as template specialisation

Syntax

```
template<>
return_type function_name(args) {
    // ...
}
```

Variadic Templates (C++11)

Variadic feature allows for argument lists of any size for template functions.

Summary of Rules

- Parameter packs can be declared for class templates as well as function templates

```
template<parms, typename... pack_name> class_declaration
template<parms, typename... pack_name> function_declaration
```

- Within a function declaration parameter pack can be used to declare a packed argument list, which comes at the end of the list of function arguments.
- the argument list can be qualified with `const` and/or reference operators
- parameter packs cannot be used to instantiate an object
- Number of elements can be determined using `sizeof...` operator

C I/O

Print/Scan Formats

- `%c`: single character
- `%d`: signed decimal integer
- `%i`: signed decimal integer
- `%e`: floating-point number (scientific notation)
- `%E`: floating-point number (scientific notation)
- `%f`: floating-point number (decimal format)
- `%G`: floating-point number that generates fewest characters (scientific, decimal format)
- `%o`: unsigned octal character
- `%s`: C-string printed until terminating null
- `%u`: unsigned decimal integer
- `%x`: unsigned hex number
- `%X`: unsigned hex except uses uppercase letters
- `%p`: pointer or address value
- `%%`: literal %

Advanced `printf` format syntax

```
% [flags] [width] [.precision] [length_char] specifier
```

Flags

- `+` +/- sign printed for signed values
- `-` left-justifies the output
- `#` octal prefix or hex prefix to be printed
- *space* space printed if no +/- sign is going to be printed
- `0` field padded with `0`s instead of spaces

Width: width of the field

Precision: number of digits printed (for floating point values)

length_char

- `h` short integer format

- `l` long integer format
- `L` long double format

`scanf` formats

- `%c` character
- `%d` digit as decimal integer
- `%f/e/E/g/G` floating point
- `%o` octal digit
- `%s` string, read until whitespace encountered
- `%u` unsigned decimal
- `%x/X` hex

`%[*][width][length_char]specifier`

asterisk indicated data is to be read but ignored

width max number of characters to be read for this field

length_char size of the data type involved

- `h` short
- `l` long
- `L` long double

Input and Output to Strings

`sprintf` and `sscanf` can be used to write to a string directly.

File I/O

1. Declare a file pointer `FILE *fp`
2. Get file pointer by calling `fopen`
3. Use file pointer to read/write to the file
4. Close the file `fclose`

Opening a file

`fopen(filename, mode`

Modes:

- `r`: read-only
- `w`: write, if exists contents are erased
- `a`: append
- `r+`: read/write, file must exist
- `w+`: read/write, if exists contents are erased
- `a+`: read/write, previous contents protected
- `t`: (def) text mode
- `b`: binary mode, can be used as a modifier on all previous modes

Closing a file

Closing a file is good practise, open files may not be accessible to other programs. Changes are realised upon closing.

`fclose(file_ptr)`

Reading and Writing Text Files

- `fputs(str, file_ptr)`: writes `str` to file
- `fgetc(file_ptr)`: returns the next character from the specified input stream
- `fgets(file_ptr)`: reads characters from input stream until a newline or EOF
- `fprintf(file_ptr, format_str, [,args])`: writes formatted output to output stream
- `fputc(ch, file_ptr)`: writes char to file
- `fputs(str, file_ptr)`: writes a C-string to file
- `fscanf(file_ptr, format_str, [,args])`: reads texts interpreted according to format specified
- `getc(file_ptr)`: returns the next character from the specified input
- `putc(ch, file_ptr)`: writes `ch` to specified file

Reading and Writing Binary Files

`reinterpret_cast<char*>` operator should be used.

Random-Access Function

Random-access functions move the file position indicator, by default the pointer moves sequentially.

- `fgetpos(file_ptr, fpos_ptr)`: saves the current file pointer into an object of type `fpos_t`
- `fsetpos(file_ptr, fpos_ptr)`: restores a saved position
- `fseek(file_ptr, offset, origin)`: sets the file position indicator to an offset from a specified origin -`ftell(file_ptr)`: returns the file-position indicator for the specified stream, returned as a `long int`

Other File-management functions

- `clearerr(file_ptr)`: clears the error status of a stream
- `ferror(file_ptr)`: returns non-zero value if there is an error
- `fflush(file_ptr)`: flushes the input or output buffer - reads and writes all the data in the buffer
- `freopen(filename_str, mode_str, file_ptr)`: attempts to close stream and reassigns the stream to the named file
- `remove(filename_str)`: deletes the named file from disk

- `rename(oldname_str, newname_str)`: renames the *oldname* to *newname*
- `rewind(file_ptr)`: reset the position indicator to the beginning of the specified stream
- `tempfile()`: returns a file-pointer that can be used as a temporary file
- `tempname(str)`: returns an available temporary file name as a C-string

Math, Time, and Other Library Functions

- `#include <cmath>`

Math

Trig

- `sin/cos/tan`
- `axxx`: inverse of `xxx`
- `xxxh`: hyperbolic `xxx`

Other

- `abs(x)`: absolute value
- `ceil(x)`: ceiling
- `exp(x)`: exponent
- `floor(x)`: floor
- `log(x)`: natural logarithm
- `log10(x)`: logarithm in base 10
- `pow(base, exponent)`: base raised to exp
- `sqrt(x)`: square root

C Data and Time

Common use cases

1. `time` function to get current time `time(NULL)` gives current time.
2. `localtime(&tm)` breaks `tm` into components
3. `strftime` display time in string format
4. `ctime` display standard timestamp

- `asctime(tm_ptr)`: pointer to a `tm` struct, returns a C-string format
- `clock()`: returns the number of internal clock ticks (~1/1000 secs) since the program bega
- `ctime(time_t_ptr)`: C-string timestamp
- `difftime(time_t_1, time_t_2)`: C-string timestamp
- `gmtime(time_t_ptr)`: produces GMT time by adjusting system timezone
- `localtime(time_t_ptr)`: returns a `tm` struct
- `mktime(tm_ptr)`: returns `time_t` val corresponding to the `tm` struct

- `strftime(dest_str, n, fmt_str, tm_str)`: writes formatted date/time to *dest_str*
- `time(time_t_ptr)`: returns the current time as a `time_t` value

`TM` data structure

`tm` struct contains the time/date information broken down into components.

Members

- `tm_sec`: seconds 0-59
- `tm_min`: minutes 0-59
- `tm_hour`: hours 0-59
- `tm_mday`: day of month 1-31
- `tm_mon`: day of month 0-11
- `tm_year`: number of years since 1900
- `tm_wday`: day of week ranging from 0-6
- `tm_yday`: day of year ranging from 0-365
- `tm_isdst`: indicates whether daylight saving time is in effect

Date/Time format Specifiers

- `%a`: day of the week, three letter abrv
- `%A`: day of the week, full
- `%b`: name of the month, three letter abrv
- `%B`: name of the month, full
- `%c`: complete data/time *mm/dd/yy hh:mm:ss*
- `%d`: day of moths as 2 digit number
- `%H`: hour as 2 digit number 00-23
- `%I`: hour as 2 digit number 00-11
- `%j`: day of year, 3 digit number
- `%m`: month as 2 digit number
- `%M`: minutes as 2 digit number
- `%p`: two char AM or PM
- `%S`: second as 2 digit number
- `%U`: week as two digit number 0-53
- `%w`: weekday as decimal
- `%W`: week as number, same a `%U` but first day is Monday
- `%x`: locale-dependent date representation
- `%X`: locale-dependent time representation
- `%y`: string with last 2 digits of year
- `%Y`: 4 digit string of year
- `%Z`: time zone

String-to-Number conversions

- `atof(s)`: digit string to double

- `atoi(s)`: digit string to int
- `atol(s)`: digit string to long
- `strtod(s, ptr_to_s)`: sets *ptr_to_s* to first char not successfully read, useful for strings with multiple digits

Memory Allocation Functions

In C++ `new` and `delete` are preferable. The advantage of the C functions is that they include `realloc` which resizes an array while preserving the contents.

- `calloc(size, count)`: attempts to allocate a memory block *size x count*. If the function succeeds `*void` returned.
- `free(ptr)`: releases memory previously allocated by `calloc` / `malloc`
- `malloc(size)`: attempts to allocate memory of block *size* large
- `realloc(ptr, new_size)`: reallocates the memory of block pointed to by *ptr* adjusting for memory block to have new size.

Standard C randomisation functions

- `rand()`: returns a pseudo-random number
- `srand()`: sets the seed for the random-number generator

Searching and Sort

- `bsearch`: searches an array for a target value, returns a pointer if found or null pointer if not
- `qsort`: sorts all the values of an array, leaving it in ascending order - uses the quick sort algorithm

Misc

- `abort()`: terminates the program
- `atexit(funct)`: registers an exit function called on termination
- `exit(n_status)`: causes the program to end normally returning status code *n_status*
- `getenv(env_var_str)`: returns the specified environment variable as a C-string
- `ldiv(n1, n2)`: division but returns as long integer
- `system(command_str)`: send *command_str* to the system

C++ I/O Stream Classes

Steam classes are more extensible, simpler to use and consistent i.e. they work the same way for console i/o as they do for file i/o

C++ I/O Streams Basics

Writing Output with <<

`cout` used to write to console. Modifiers can be used to change how data is presented

```
cout << modifier << ...
```

Stream manipulators

- `hex`: change format to hex

Reading Input >>

`cin` is used to read data in. Objects received my be *lvalues*, except in the case of C-strings which are addresses.

For strings, input is read up to and not including the first whitespace.

Limitations:

- `cin`: no input appears until the user presses enter
- non reliable input causes the program to freeze
- no more reads are possible until the error flag is cleared
- leading white-spaces are ignored

`>>` should mainly be used in simple cases

Reading Line Input with `getline`

Two major versions of `getline`

- `istream_obh.getline(cstr, n [, delim_ch = '\n'])`: reads line input into a C-string
- `getline(istream_obj, str_obj [, delim_ch = '\n'])`: reads a line of input into a `string` object

`getline` member to read input into a C-string. `getline` global function to read a line of input into a `string` object.

To interpret complex input, input can be tokenised with `strtok`

C++ Stream Class Hierarchy

`cout` object is an instance of the `ostream` class - derived from the base class `ios`.

Stream output objects are instances of `ostream`, or in the case of file output, `ofstream`. Input is derived from `istream` and `ifstream`.

Stream objects include

- `cout`: Console output, instance of `ostream`, can be written to with the `<<` operator.
- `cin`: Console input, instance of `istream`. Input is not available until `Enter` is pressed. Stream input operator `>>` skips over leading spaces, errors can be tested in a `while` loop
- `cerr`: Console error output, instance of `ostream`, can be redirected to any file.
- `clog`: Console log output. Instance of `ostream`

Stream Objects: Manipulators and Flags

Many ways to control and format output - `<iostream>` header gives access to most predefined objects, `<iomanip>` provides access to `set` and `reset` manipulators.

Stream Manipulators

Manipulators modify a stream in some way - they change the format of subsequent operations on the stream until the format is reset.

Manipulators fall into a number of categories:

- Manipulators that set the base
- Manipulators that set flags
- Manipulators that set the width, fill and precision
- Affect floating point behaviour
- Control print justification
- Termination `endl`, `ends`

Stream manipulators:

- `boolalpha`: causes true/false to be printed as `true` / `false`
- `dec`: Decimal format for integers
- `endl`: prints a newline character
- `ends`: prints a null character
- `fixed`: fixed point format fro floating point integers
- `flush`: flushes the buffer associated with the stream
- `hex`: specifies hex format for integers
- `internal`: sets print-field justification to internal rather than left / right
- `left`: print fields are left-justified
- `noboolalpha`: turns off `bools` printed as `true` / `false`
- `noshowbase`: turns off `showbase` setting, octal and hex prefixes are not printed
- `noshowpoint`: turns off `showpoint` flag, floating point numbers are not forced to be printed with a `.`
- `noshowpos`: turns off `showpos` flag positive signed numbers are not displayed with a `+`
- `noskipws`: turns off `skipws` flag, leading spaces are not skipped

- `nounitbuf`: turns off `unitbuf` flag, which causes stream to be flushed after every read / write
- `nouppercase`: turns off hex and exponents being printed in uppercase letters
- `oct`: octal as integer format
- `resetiosflags(bitmask)`: clears the format flags indicated by the *bitmask* arg
- `right`: print fields are right-justified
- `scientific`: scientific format for floating-point numbers, so exp portion is always displayed
- `setfill(ch)`: sets the fill character to *ch*
- `setiosflags(bitmask)`: sets the format flags indicated by *bitmask* arg
- `setprecision(n)`: Set precision of floating pint numbers to *n* - `showpoint` in effect, precision is total digits to left and right, `scientific` or `fixed`, precision is total digits to the right
- `setw(n)`: sets minimum print field with to *n*
- `showbase`: hex and octal numbers are printed with prefix
- `showpoint`: floating point numbers are printed with the decimal point even if they are whole numbers
- `showpos`: positive numbers are printed with a leading `+`
- `skipws`: leading whitespaces are skipped during stream input operations
- `unitbuf`: Output buffer flushed on each write operation
- `uppercase`: hex and exp printed in uppercase
- `ws`: Reads an ignore whitespaces until non-whitespace character found

Stream Format Flags

Stream flags are used with `setiosflags` and `resetiosflags`, the bitwise OR operator `|` can be used to combine flags

```
cout << setiosflags(ios_base::fixed | ios_base::showpoint);
```

Flags

- `ios_base::boolalpha`: displays `true`/`false` bool values as "true" and "false"
- `ios_base::dec`: display and read integers as decimal format
- `ios_base::fixed`: display fixed floating point format
- `ios_base::hex`: display and read integers as hex
- `ios_base::internal`: internal justifying for printing
- `ios_base::left`: left-justifying in print fields
- `ios_base::oct`: display and read integers as octal
- `ios_base::right`: right-justify fields
- `ios_base::scientific`: display floating points values in scientific format
- `ios_base::shwobase`: prefix hex and octal
- `ios_base::showpoint`: specifies that floating point number should be printed

- `ios_base::showpos`: display `+` for positive numbers
- `ios_base::skipws`: prevents an input stream from skipping leading whitespace
- `ios_base::unitbuf`: causes input buffer to be flushed with each read / write op
- `ios_base::uppercase`: displays hex prefix and digits as uppercase

Steam Member Functions

Input stream functions (`cin`)

- `istream_obj.gcount()`: number of characters rest by the last call of `getline` or `get`
- `istream_obj.get()`: gets the next character from the input buffer and returns it as an integer
- `istream_obj.get(ch)`: gets next char and puts it in *ch*
- `istream_obj.get(cstr, n [, delim])`: gets up to *n* chars stopping at *delim*
- `istream_obj.getline(cstr, n [, delim])`: reads input into C-string.
- `istream_obj.ignore([n = 1] [, delim = EOF])`: reads up to *n* characters or until the delim is read, and ignores them
- `istream_obj.peak()`: returns the next character in the input buffer without removing it from the buffer, enables look ahead functionality
- `istream_obj.putback(ch)`: Puts a character back into the input buffer, the position counter is decremented by 1 and *ch* is the next character to be read

Output Stream Functions

- `ostream_obj.put(ch)`: puts a character *ch* onto the output stream
- `ostream_obj.fill(ch)`: changes the fill character to *ch*
- `ostream_obj.flush()`: flushed the output buffer so that data in the buffer is written to the file
- `ostream_obj.precision(n)`: sets floating point precision to *n*
- `ostream_obj.width(n)`: specifies the minimum print-field width

Flag-Setting Stream Functions

Functions are a way to set format flags

- `stream_obj.bad()`: returns `true` if bad bit is set, indicating an error
- `stream_obj.clear()`: clears the error flag
- `stream_obj.copyfmt(stream)`: copies the format flag setting from another stream
- `stream_obj.fail()`: returns `true` if the fail bit or bad bit is set, indicating an error

- `stream_obj.flags()`: returns the current state of the format flags as an integer
- `stream_obj.flags(bitmask)`: sets the flag values according to the values in *bitmask*
- `stream_obj.good()`: returns `true` if the "good" bit is set for the file stream
- `stream_obj.setf(bitmask)`: turns on the flags indicated by *bitmask*
- `stream_obj.setf(bitmask1, bitmask2)`: only the flags specified by *bitmask2* are changed
- `stream_obj.unsetf(bitmask)`: switches off the flags indicated by *bitmask*
- `stream_obj.sync_with_stdio([on/off = true])`: turns on sync between stream classes and the C standard IO functions

File Stream Operations

C++ stream classes support both text and binary file I/O. Binary files are more efficient as data does not need to be encoded, and therefore copied directly.

Creating a File Object

`<fstream>` header is required.

- `ifstream`: object for file input
- `ofstream`: object for file output

Mode flags

- `ios_base::app`: Append mode, if the file already exists open it in append mode, otherwise create the file
- `ios_base::ate`: At-End mode, the files position indicator is moved to the end of the file, needs to be combined with `in`, `out`, `app`
- `ios_base::binary`: Binary mode, no translation of newlines is performed
- `ios_base::in`: Input mode (default for `ifstream`), the file must already exist
- `ios_base::out`: Output mode, (default for `ofstream`), if the file does not exist it is created
- `ios_base::trunc`: if the file exists, file contents are erased

File-specific member functions

- `fstream_obj.close()`: closes the file associated with the stream object
- `fstream_obj.eof()`: returns true if the end-of-file condition is detected
- `fstream_obj.is_open()`: returns true if the stream object has a file associated with it open for reading and writing
- `fstream_obj.open(filename [, mode])`: opens a file, function returns a reference to the stream itself

Reading and Writing in Binary mode

- `istream.read(p_data, size)`: Reads *size* bites directly from the stream and stores it at the address specified by *p_data*, a pointer of type `char*`
- `ostream.write(p_data, size)`: writes *size* bites from the address specified by *p_data* directly to the stream.

Random-Access Operations

- `istream.seekg(n [, ref_point])`: moves the stream position indicator to *n* where *n* is the offset, *ref_point* is the initial reference point
- `ostream.seekp(n [, ref_point])`: moves the stream position to *n*, where *n* is the offset, *ref_point* is the initial reference point
- `istream.tellg()`: returns the integer of type `streamoff` containing the current position indicator
- `ostream.tellp()`: returns the integer of type `streamoff` containing the current position indicator

Reading and Writing String Streams

A string stream allows you to leverage all the tools of formatted I/O to strings i.e. it is much easier to write formatted output to a `stringstream` object

```
#include <string>
#include <sstream>
// ....
stringstream s_out;
int n = 255;
s_out << "The value of n in hex is: << hex << showbase << n << endl;
```

- `stringstream name( [mode])`: `stringstream` constructor, initialised with empty string
- `stringstream name(str, [mode])`: `stringstream` constructor, initialised with the contents of *str*
- `stringstream_obj.str()`: returns a `string` objects containing the current text data stored
- `stringstream_obj.str(str)`: copies text data to *str* reference

Overloading Shift Operators

```
ostream& operator<< (ostream &os, const my_class &cls) P
    os << // ...
    return os;
```

STL Algorithms

Much of programming come down to sorting, searching, counting and selecting. The STL algorithms are function templates that perform general functions on nearly any kind of data.

General Concepts

Generally the `<algorithm>` header is required to get started, some other algorithms are under the `<numeric>` header.

Arguments usually include iterators that point to the beginning and end positions within a container, referred to as *beg* and *end*. Containers must provide their own iterator.

```
#include <algorithm>
// ...
int arr[8] = {100, 1234, 4, 18, 1, 89, 43, 2};
sort(arr, arr + 8);
```

Algorithms that take functions as arguments can be passed them as:

- The address
- A function object
- A lambda function

STL container bounds can always be determined.

Using Lambda Functions (C++11)

```
[] (int n) { cout << n; }
```

Lambdas also let you capture variables from the surrounding scope. For example

```
int sum = 0;
[&sum] (int n) {
    sum += n;
    cout << n << " ";
}
```

The return type can be specified via

```
[] (args) -> return_type { //... }
```

Algorithms and Iterators

Each algorithm works with a specific type of iterator - random access iterators are supported by all algorithms.

Insert Iterators

Are used primarily with STL algorithms and STL containers. To use them include the `<iterator>` header.

Iterator functions include

- `back_inserter(con)`: calls `con.push_back(val)` and supported by most STL containers but not arrays
- `front_inserter(con)`: calls `con.push_front(val)` and supported by most STL containers but not vectors
- `inserter(con)`: calls `con.insert(val)` and supported by most STL containers

Summary of Algorithms

STL supports many algorithms, broadly grouped into:

- Read-only
- Modifying
- Sorting
- Heap (specialised)
- Numeric, which apply a mathematical operation

Summary Read-only

- `adjacent_find`: finds the first occurrence of two consecutive elements having the same value
- `all_of, any_of, none_of`: tests a container if all, any, or none of the elements fulfil a specified condition
- `binary_search`: searches an ordered container for a specified value
- `count, count_if`: returns the number of elements matching a specified value or condition
- `equal`: compares two containers or sub-ranges for equality, i.e. each corresponding element must be equal
- `for_each`: perform a specified action for each element
- `find, find_if`: find the first element matching a specified value or condition
- `find_if_not`: finds the first element not fulfilling a specified condition
- `find_end`: finds the last occurrence of a subrange within a larger range
- `find_first_of`: find the first element matching any element that is also an element of a specified range
- `includes`: determines whether *every* element in a range matches some element in another container or subrange
- `lower_bound, upper_bound, equal_range`: returns the iterators representing the lower and upper bounds for a range specified
- `max`: returns the larger of two values
- `min`: returns the smaller of two values

- `max_element, min_element`: find the position of the max / min element
- `minmax_element`: returns the positions of the min and max element
- `mismatch`: compares two ranges and reports the first mismatch
- `search_n`: search fora series of consecutive elements all meeting some condition
- `search`: finds the first occurrence of a complete subrange within another range

Modifying Algorithms

- `copy, copy_backward`: copy elements in a range into a destination range
- `copy_n, copy_if`: copy n elements in a range or if a condition is met
- `fill, fill_n`: set all the elements in a specified range to a particular value
- `generate, generate_n`: set the elements in a range by calling a generator function
- `iter_swap`: swaps two elements
- `merge, inplace_merge`: merge two sorted ranges together into a single sorted range
- `move, move_backward`: move all the elements of one range into another, not a copy action - possibly destructive
- `remove, remove_if, remove_copy, remove_copy_if`: remove one or more elements that meet a specific condition
- `replace, replace_if, replace_copy`: replace the value of one or more elements that meet a specific condition
- `swap`: exchanges any two values having the same base type
- `swap_ranges`: swaps two ranges of elements
- `transform`: performs an operation on a range of elements and places the result in a destination

Sorting and Re-ordering algorithms

- `is_permutation`: compares two ranges and checks if one is a permutation of the other
- `is_sorted, is_sorted_until`: Returns `true` if the range is already sorted - or the position of the element breaking the sort
- `partial_sort, partial_sort_copy`: sort part of a range
- `is_partitioned`: returns `true` if the range is ordered as a partition
- `partition_point`: find the element that divides the contained into a partition
- `prev_permutation, next_permutation`: rearrange elements so that they form the previous or next permutation
- `random_shuffle`: randomly shuffle the contents of a container or subrange
- `reverse, reverse_copy`: rearrange a series of elements so that their order is the reverse
- `rotate, rotate_copy`: move elements forward or backward in a container

through rotation
- `set_difference`: finds the difference between two ordered ranges
- `set)intersetion`: creates the intersection between two ordered ranges copying the result to a destination range
- `set_symmetric_difference`: finds the symmetric difference between two ordered ranges
- `set_union`: creates a union between two ordered ranges, copying the result to a destination range
- `sort`: sorts a container or subrange
- `stable_sort`: sort a container preserving the relative position of elements with equal values
- `unique, unique_copy`: remove consecutive elements having the same value

Heap Algorithms

The term "heap" is used in a specialised way, these algorithms reorder a random-access container (array, vector, deque) to treat it as a virtual top-down binary tree.

The "highest" element is the element at the root of the tree. Each child of a node is less than or equal to the node itself

Elements in a heap can be inserted more quickly into larger collections than they can with linear searches. The heap search time grows logarithmically, i.e. search a container with millions of elements takes twice as long as it does for a collection with thousand

- `make_heap`: reorders a range within a random access container so that it fulfils the condition of being a heap
- `is_heap`: returns `true` if the range is a arranged as a heap
- `is_heap_until`: returns the first position of the container that breaks the requirements of being a heap
- `pop_heap`: removes the first element of the heap and rearranges the remaining elements s.t. they are a heap
- `push_heap`: insert a new element into a heap
- `sort_heap`: sorts a range ordered as a heap

Numeric Algorithms

The numeric algorithm are part of the `<numeric>` header, they involve mathematical manipulation or calculation.

- `accumulate`: adds up the contents of a range and returns the result
- `adjacent_difference`: finds the difference between adjacent elements
- `intter_product`: accumulates the inner product
- `iota`: sets the first element in a range to a specified value and then each other element to 1 plus the previous value

- `partial_sum`: for each element find the partial sum formed by adding it to all preceding elements

Numeric algorithms work with functions and function objects. There are a number of templates that can return function objects performing addition, subtraction and so on. These templates require the `<functional>` header. These are useful in confection with numeric algorithms

- `plus<type>()`: returns the function object that adds together its two arguments
- `minus<type>()`: returns the function object that subtracts the second arg from the first
- `negate<type>()`: returns the function object that produces the arithmetic negation
- `multiplies<type>()`: returns the function object that multiplies its two arguments
- `divides<type>()`: returns the function object that divides the first arguments by the second
- `modulus<type>()`: returns the function object that performs the modulus
- `equal_to<type>()`: returns the function object that tests two args for equality
- `less<type>()`: returns the function object that performs a < b
- `greater<type>()`: returns the function object that performs a > b
- `less_equal<type>()`: returns the function object that performs a <= b
- `greater_equal<type>()`: returns the function object that performs a >= b
- `logical_not<type>()`: returns the function object that produces the logical negation
- `logical_and<type>()`: returns the function object that produces the logical AND
- `logical_or<type>()`: returns the function object that produces the logical OR
- `bit_and<type>()`: returns the function object that performs the bitwise AND
- `bit_or<type>()`: returns the function object that performs the bitwise OR
- `bit_nor<type>()`: returns the function object that performs the bitwise exclusive OR

C++11 Randomisation Library

The C+11 specification provides more reliable, versatile and less biased random number generators.

Issues with Randomisation

Problems with randomisation include

- biased distributions
- pseudo-random sequences
- getting a seed

A Better Scheme

When using the C++11 randomisation you need to:

1. Choose an engine that generates raw pseudo-random numbers, some engines provide better simulations of randomness than others
2. Choose an initial seed (unless you're debugging)
3. Choose a distribution, the STL randomisations are optimised to produce numbers in a desired range as efficiently as possible

To use: `#include <random>`, all symbols are part of the `std` namespace

```
default_random_engine eng(seed); // randomisation engine
uniform_int_distribution<int> dist(1, 6); // distribution
```

The same engine should be kept in use. Starting a new engine might be problematic as it may give the same sequence of values (unless over a second has past).

The seed should be changed if a new randomisation instance is going to be created.

Common Engines

There are a wide choice of engines in the C++11 randomisation library. The most commonly used engines are:

- `default_random_engine`: default engine for randomisation - choice is implementation dependent, so may get a different engine on a different platform
- `minstd_rand`: version of the `linear_congruential_engine`, fast and takes up little space in memory - small state and liable to repeat
- `minstd_rand0`: similar to `minstd_rand`, usually just as fast but with slightly better results
- `ranlux24_base`: version of `subract_with_carry` engine, reasonable speed but takes up more memory than `minstd_rand`
- `mt19937`: version of the `mersenne_twister` engine, high-quality, high-speed and produces a sequence 2^19937 - 1 long before repeating, takes up more space in memory than other engines, seed value needs to be picked carefully.
- `mt19937_64`: similar to `mt19937` but operates in 64 bits

- `knuth_b`: complex adaptation that applies the `shuffle_order_engine`, high quality but slower than other engines

Common Distributions

- `uniform_int_distribution<type> name(first, last)`: discrete uniform between *first* and *last* of type *type* (`short`, `int`, `long`)
- `uniform_real_distribution<type> name(lower, upper)`: uniform distribution of floating point numbers usually `double` or `float`, where x in [*lower*, *upper*)
- `bernoulli_distribution name(prob=0.5)`: Bernoulli distribution, produces `true` / `false` with probability *prob*
- `binomial_distribution<type1, type2> name(n, prob)`: produces sum of *n* Bernoulli trial, *type1* and *type2* are the types of *n* and *prob* usually `int` and `double`
- `exponential_distribution<type> name(rate)`: Exponential distribution, produces floating point numbers of specified *type*, usually `double`
- `geometric_distribution<type1, type2> name(prob)`: Returns an integer of specified *type1* based on a *prob* of specified *type2*
- `normal_distribution<type> name(mean = 0.0, std_dev = 1.0)`: Produces a floating point number from the Normal distribution
- `poisson_distribution<type> name(rate)`: Takes a *rate* usually `double` and produces an integer

There are many more distributions including `gamma`, `weibull`, `extreme_value`, `chi_squared`, `cauchy`, `fisher_f`, `student_t`, `discrete`, `piecewise_constant`, `piecewise_linear`

Operations on Engines

- `engine.seed()`: Resets an engine to its default initial state
- `engine.seed(seed_val)`: Sets a new seed for an existing engine
- `engine()`: Gets raw output from the engine
- `engine.discard(n)`: Advances random number sequence *n* steps
- `ostream_obj << engine / istream_obj >> engine`: Performs serialisation, read and write the internal value of the engine in text form (i.e. save and reset the state).

Operations on Distributions

- `dist(engine)`: produces a pseudo-random number from the distribution
- `dist.min()`: returns the minimum number the distribution can produce
- `dist.max()`: returns the maximum number the distribution can produce
- `ostream_obj << dist / istream_obj >> dist`: Performs serialisation, read and write the internal value of the distribution in text form (i.e. save and reset the state).