

# C++ Advanced

*Advanced Techniques and Concepts  
for C++ Programmers*

---

Curbralan Ltd  
<http://curbralan.com>

---

## Contents

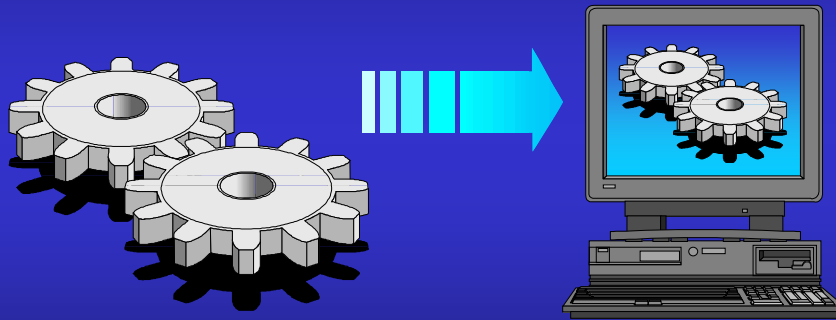
- |                                |                                 |
|--------------------------------|---------------------------------|
| 1. Course Introduction         | 10. Function Objects            |
| 2. C++ Review                  | 11. Exception Safety            |
| 3. Conversions and Mutability  | 12. Dynamic Resource Management |
| 4. Scope                       | 13. Dependency Management       |
| 5. Value-Based Objects         | 14. Policy-Based Design Basics  |
| 6. Towards Generic Programming | 15. Course Outroduction         |
| 7. Containers                  | 16. Appendix: C++11             |
| 8. Iterators                   | 17. Appendix: Labs              |
| 9. Encapsulated Algorithms     |                                 |

*C++ Advanced*, version 14110.

© Curbralan Limited, 2014, <http://www.curbralan.com>.

Curbralan Limited asserts its moral right to be regarded as the author of this material, all rights reserved. Curbralan Limited owns all copyright and other intellectual property rights associated with this course material. No part of this publication may be stored, reproduced or transmitted in any form or by any means without the prior written permission of the copyright owner. Curbralan Limited shall not be liable in any way in respect of any loss, damages or costs suffered by the user, whether directly or indirectly as a result of the content, format, presentation or any other use or aspect of the materials.

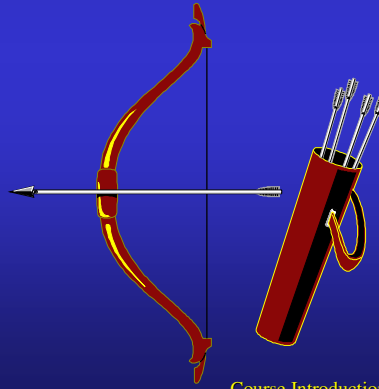
# Course Introduction



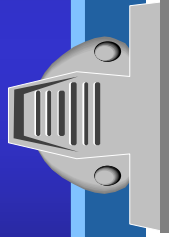
*C++ Advanced*

# Course Introduction

- Objectives
  - ◆ Define the scope and purpose of the course
- Contents
  - ◆ Course objectives
  - ◆ Course prerequisites



# Course Objectives



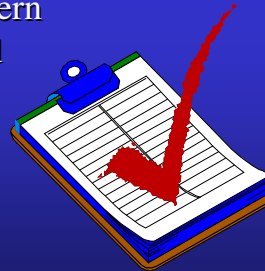
- **Build on C++ knowledge to further introduce the language**
- **Present the key features of the standard library**
- **Practise and extend OO and generic programming skills**
- **Emphasise good practice and safe idioms**

ISO standard C++98 is a general purpose language that bridges different styles of programming — from procedural, to object-oriented and generic programming with templates — and spans different platforms and different application styles — from embedded systems through to GUI applications.

This course provides developers with an armoury of modern C++ techniques and concepts suitable for serious development, covering the various styles C++ supports as a language and a library — procedural, object-based, generic and object-oriented programming. It develops the concepts and syntax through lectures, discussion and hands-on lab exercises.

## Course Prerequisites

- Essential...
  - ◆ Good C++ programming skills
- Useful...
  - ◆ Experience of using C++ standard library
  - ◆ Some previous exposure to modern OO development techniques and concepts, such as design patterns



The course is suitable for C++ programmers, but not for those who are new to the language. Experience of the standard library and of concepts such as design patterns and test-driven development (TDD) is useful but not essential.

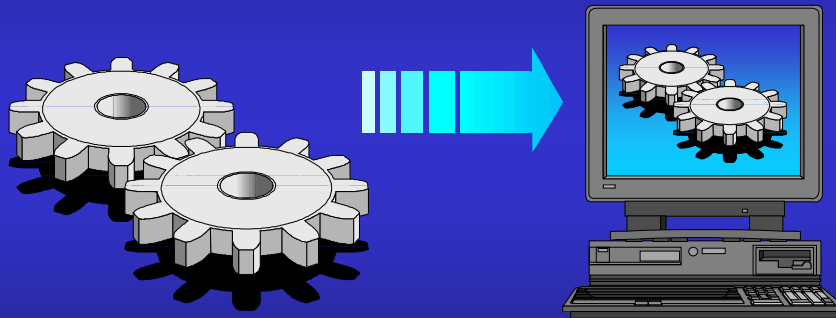
If you are unsure about meeting the prerequisites, please speak to the instructor.

## Any Questions?

- Please feel free to ask questions at any time
  - ♦ The surest way of having your questions answered is to ask them!



# C++ Review

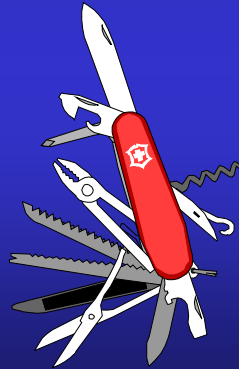


*C++ Advanced*



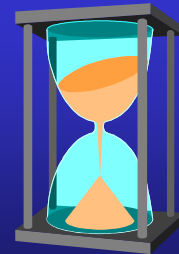
# C++ Review

- Objectives
  - ♦ Provide a C++ language and library refresher
- Contents
  - ♦ Characterising modern C++
  - ♦ Classes and derivation
  - ♦ Operator overloading
  - ♦ Exception handling
  - ♦ *new* and *delete*
  - ♦ The standard library and the STL
  - ♦ Beyond the standard library



## The Ages of C++

- C++ has evolved through reasonably distinct phases
  - ♦ Early C++ → Classic C++ → Modern C++
  - ♦ Each phase corresponds roughly with editions of *The C++ Programming Language*
- Each era can be regarded as a separate language with different features and styles
  - ♦ The new era, based on the C++11/14 standard, also promises to be distinct



C++ has evolved through three reasonably distinct phases, each phase corresponding roughly with editions of *The C++ Programming Language*. The finalisation of the standard with its library brought C++ into its third era: the first being the early adopter phase from release in 1985 to around 1990, and the second was heralded in 1991 by the successful publication of a number of good quality practitioner books. The third era sees a new style for C++ use quite different from the early C-like or Smalltalk-clone attempts at using the language. Each age can be regarded as a separate language with different features and supporting different styles: Early C++, Classic C++ and Modern C++.

These three eras are not separated by crisp boundaries; as with any evolving language, each era fades into the next, the seeds of subsequent styles and demand for features are often sown in the previous. For instance, although the STL was made available in 1994 and then adopted into the draft standard, its full impact on design style was not felt and understood until later.

# Characterising Modern C++

Static scoping and type checking

Classes

Inheritance

Runtime inclusion polymorphism

Overloading

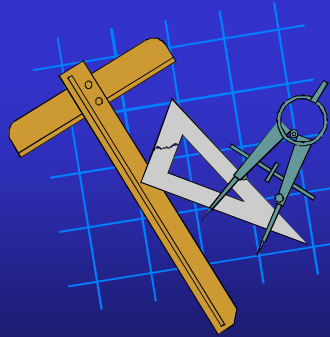
Templates

Value-based object semantics

Manual heap management

Exceptions

*A hybrid language  
with support for  
multiple paradigms*



© Curbralan Ltd

C++ Review 11

The degree to which a language or technology specifically supports certain mechanisms will have an impact not only on the way that programmers will think about a problem, but also on the way that an application should be designed.

Here are a few of C++'s features that are likely to have an impact:

- Static scoping and type checking: a static type system, plus conversion mechanisms.
- Classes: user-defined types and class-level encapsulation, supporting information hiding and member functions.
- Inheritance: abstract and concrete classes, single or multiple inheritance of implementation, and `public`, `protected` and `private` inheritance.
- Runtime inclusion polymorphism: static versus dynamic binding, substitutability of derived classes for base classes.
- Overloading: overloading of function names and operators.
- Value-based object semantics: copy semantics and deterministic object lifetime, especially useful for fine-grained, non-polymorphic objects.
- Templates: generic types and functions based on operational usage.
- Manual heap management: deterministic but manually managed lifetime for indirect (as opposed to value) objects.
- Exceptions: discontinuous transfer of control flow and data.

## Multi-Paradigm C++

- Because of its extensive feature set C++ supports multiple programming styles
  - ♦ Procedural and modular programming
  - ♦ ADT and value-based programming
  - ♦ Classic OO and component-based programming
  - ♦ Generic programming
- C++ is best appreciated in terms of these different styles rather than one



C++ is often characterised, a little negatively, as a hybrid language, or more positively as a multi-paradigm language. Either way, it supports a wide range of implementation styles, leaving the choice of paradigm for a particular problem as a design choice for the programmer.

Procedural programming is perhaps the most commonly recognised, long-standing programming paradigm, embodied as it is in classic and modern languages alike. The focus of a program is considered to be a task, which can be decomposed into subtasks, and so on. Tasks, or operations, are mapped to procedures (or functions, in the case of C and C++), and connected through a function call hierarchy. Data, in the procedural model, is secondary to procedure, and flows from function to function via arguments (or, in the worst case, global variables).

Abstract data type programming takes the view that a program is organised around types and the operations available on the types, rather than on functions and data flow. The focus on the interface to ADT instances, but not on features such as polymorphism and inheritance, often leads to ADT approaches being termed object-based rather than object-oriented.

Full object orientation is commonly based on the compile-time partitioning of a program into classes, and the runtime realisation of a program as a set of collaborating objects. OO is often said to be based on three basic mechanisms: encapsulation, polymorphism and inheritance.

# Classes

- Classes extend the ability of programmers to work with user-defined types
  - ♦ Commonly the combination of data and function
  - ♦ New classes can be written for new applications and existing classes used for established solutions

```
class url
{
public:
    std::string protocol() const;
    std::string resource() const;
    ...
private:
    std::string text;
};
```

Definition of new type → `class url`

Directly available public functions on type → `std::string protocol() const;`  
→ `std::string resource() const;`  
→ `...`

Private data representation for type → `std::string text;`

Classes allow programmers to extend the type system with types appropriate to their problem and solution domains. Classes can be defined anew or may be used from an existing library or another part of a project. Classes define both a data structure and the operations on it.

The typical style for class use is to make the key operations public and available whilst hiding the data representation. However, this is not the only approach, and some classes represent stateless objects or structures with exposed data — the fact that they are syntactically related to `structs` is no coincidence.

# Abstract Classes

- An abstract class is one that has one or more pure *virtual* functions
  - ♦ Useful for providing incomplete implementations from which classes may derive
  - ♦ Fully abstract classes offer an idiom for flexibility and loose coupling

```
class file
{
public:
    virtual bool read(std::string & to, std::size_t max) = 0;
    virtual bool write(const std::string & from) = 0;
    virtual bool eof() const = 0;
    ...
};
```

From a usage perspective, an abstract class is used as an ordinary class. The only difference is that it cannot be instantiated, so it cannot appear as the class name in a `new` expression or as the value type in a variable declaration. The most common idiom for using abstract classes is via pointers. The `= 0` on the end of a `virtual` member function declaration is what defines it to be pure `virtual`.

An "interface class" is one that has no data and the only ordinary member functions are declared `public` and pure `virtual`.

## Derived Classes

- A derived class can override any *virtual* functions in the base class(es)
  - ♦ If concrete (i.e. non-abstract) it can be instantiated
  - ♦ The *virtual* keyword is not required for overriding

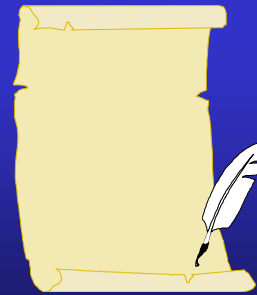
```
class stdio_file : public file
{
public:
    explicit stdio_file(const std::string & name);
    virtual bool read(std::string & to, size_t max);
    virtual bool write(const std::string & from);
    virtual bool eof() const;
    ...
private:
    std::FILE * stream;
};
```

A derived class can derive from an abstract or a concrete class, although it is generally best if concrete inheritance is avoided. The `public` keyword following the colon is significant as it indicates that the derivation is a public policy rather than a private detail. `private` derivation is rarely of use, and so is normally an oversight.

The overridden signature of any member function is the same as the one declared in the base, including any `const` qualification, but obviously excluding the pure specifier, `= 0`.

## Inheritance Guidelines

- Avoid inheritance that is solely for code reuse
  - ♦ Use inheritance to represent substitutability
- Use *virtual* functions in preference to explicitly querying an object's type
- Use interface (fully abstract) classes at the root of a hierarchy
  - ♦ Try to avoid inheriting from concrete classes...
  - ♦ Especially multiple inheritance of concrete classes



Classes that are not intended to be used as base classes should not have any `virtual` member functions — and certainly no `virtual` destructor — and should have no `protected` members. These two features send out a strong signal that a class is designed for inheritance, which is less often the case than programmers new to C++ assume.



# Operator Overloading

- Existing operators can be overloaded for new user-defined types
  - ♦ Such operators are functions with special names
  - ♦ Supports idiomatic usage for simple data types

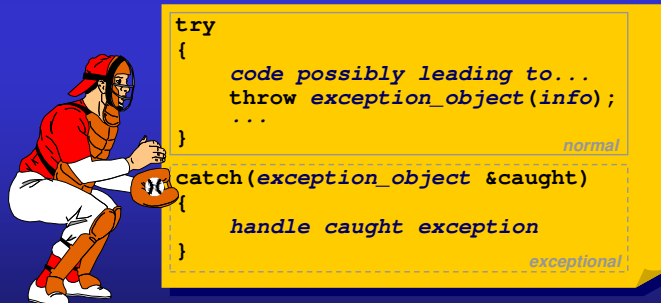
```
int main(int argc, char * argv[])
{
    const std::string spacer = " ";
    std::string args;
    for(size_t arg = 1; arg != argc; ++arg)
        args += argv[arg] + spacer;
    ...
}
```

Operator overloading is a natural extension of function overloading. Instead of always dealing with alphanumeric function names, it may be appropriate for a programmer to use one of the operators already defined in the language and overload it to work with new types. Note that there are constraints on this capability: only existing operators may be overloaded and the meaning of operators for existing built-in types may not be changed.

Of course, such a feature may be misused to create obscure code, and this can be a problem if programmers are not aware of the guidelines for appropriate use. However, obscure code can be achieved via many routes, and operator overloading should not be held responsible for poor abstraction skills. The case for operator overloading is easily made when it is used to support natural idioms for new types. In the library is used to support string concatenation, I/O streaming and numerous other features on library types.

# Exception Handling

- C++ supports separation of normal from exceptional code
  - ♦ Unification of control flow and object model



The nature of C++'s exception-handling mechanism is that it separates the "normal" code from the exception-handling code. The paths of execution are different between the two, and this helps the programmer express them as separate concepts, and to keep this distinction in the code.

Exceptions bridge the gap between the library designer and the library user. When an error condition is detected, and the library function cannot cope with it, it *throws* the exception out to a function higher in the call chain. This function can *catch* the exception and act on it appropriately – perhaps bailing out, logging the error, or attempting a retry.

Exceptions are best represented as proper objects. As objects they can be self describing, whereas something like a simple integer value – as used in other languages – is not very informative as it says nothing about the details of an exception. The use of objects to represent exceptions to some extent unifies the concept of control flow with C++'s object model.

The code that is to be attempted is placed within a `try` block. An exception arises as a result of a `throw` expression, which may be either explicit within the `try` block or present in the call chain from within it. The call chain is unwound until a `catch` – associated with a `try` – matches the exception type (either directly or as its base). The `catch` block is executed and the exception is considered handled.

## *new and delete*

- The *new* and *delete* operators provide type-safe allocation and deallocation
  - ♦ They may also be customised globally or per user-defined type
- A *new* expression is written in terms of the type to be allocated
  - ♦ On failure it throws *std::bad\_alloc*

```
traffic_light * light = new traffic_light;  
... // use light  
delete light;
```

C++ makes dynamic memory management a proper part of the language, not just the library. The `new` and `delete` operators pay attention to the types they are dealing with, in particular programmers using a `new` expression do not have to concern themselves with manually taking the `sizeof` of the right type.

## *new[]* and *delete[]*

- *new[]* and *delete[]* forms exist for allocating arrays of objects
  - ♦ However, *std::vector* is safer, better encapsulated and covers most uses of *new[]* and *delete[]*
  - ♦ Note that there is no *renew* operator — again, *std::vector* provides the functionality for resizing

```
size_t size;  
... // assign to size  
double * data = new double[size];  
... // use data  
delete[] data;
```

Dynamically allocated arrays are also accommodated in C++'s type-safe memory management system:

```
std::size_t size = ...;  
point * points = new point[size];  
...  
delete[] points;
```

The *new[]* must be matched with a corresponding *delete[]*, and not a plain *delete*. However, *std::vector* is generally an improvement on even this facility:

```
std::size_t size = ...;  
std::vector<point> points(size);  
...
```

An even simpler and safer solution.

# The Miseducation of C++

- 'Bottom C++' versus 'Top C++'
  - ◆ A low-level language for writing components versus a high-level one for using them
- Teaching the wrong C++
  - ◆ Just as a better C, along with C-based idioms
  - ◆ Just a plain, lowest common denominator OO
  - ◆ Classic C++ (or even Early C++)



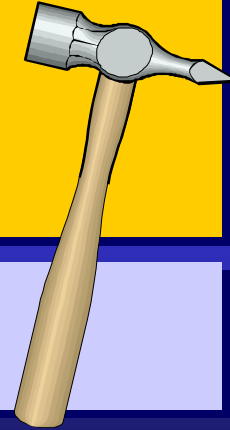
Much C++ suffers from being low level and repetitive. Whether it's the manual management of strings and arrays or the tedious transcription of similar loops, there is a lot of repetition and little abstraction of small housekeeping tasks. In short, error-prone code and a failure to reuse what exists already.

A sentiment has been growing for some time that to teach C++ from scratch should not involve a descent into memory management to achieve simple tasks. Allan Vermeulen once made a distinction that C++ was effectively two languages, one for building components and the other for using them, Bottom C++ and Top C++, respectively referring to their levels of abstraction. Using an efficient library that wraps up low-level or common facilities is quite a different prospect to writing one, and each does not require the same type or level of knowledge. This separation of views between clients and suppliers is one that has been preached for a long time in OO thinking.

## C++ as a Low-Level Language

```
char * full_name(const char * first, const char * last)
{
    const size_t length =
        strlen(first) + strlen(last) + 1;
    char * result = new char[length + 1];
    strcpy(result, first);
    strcat(result, " ");
    strcat(result, last);
    return result;
}
```

```
{
    char * name = full_name(first, last);
    ... // perform task using name
    delete[] name;
}
```



Text manipulation and good ol' fashioned I/O have come back into fashion thanks to HTML, XML and e-commerce in general, so it's worth seeing the difference between introducing C++ with and without the standard library in this context. Consider the common task of concatenating two given strings with a space between them, e.g. a person's first and last name. This task is not exactly rocket science, but with a nuts-and-bolts view of C++ this looks C-like and something like the code above.

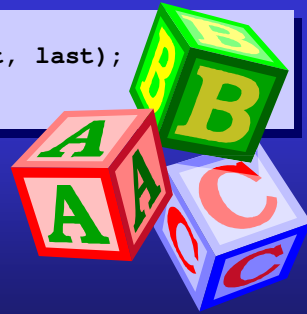
It is also possible to write the `full_name` function in a single `return` statement, but it is unlikely anyone would thank you for it. Using the function also demands a certain amount of hoop jumping. However, even in this error-prone form it is still not quite correct: it is not exception safe. To address the possibility of premature block exit from a thrown exception, you would need the following usage code:

```
{
    char * name = full_name(first, last);
    try
    {
        ... // perform tasks using name
    }
    catch(...)
    {
        delete[] name;
        throw; // repropagate exception to caller
    }
}
```

## C++ as a High-Level Language

```
using std::string;
string full_name(const string & first, const string & last)
{
    return first + " " + last;
}
```

```
{
    std::string name = full_name(first, last);
    ... // perform task using name
}
```



The library-based approach is somewhat easier to understand, to work with, and to get right. It seems odd that anyone would consider the previous approach as being better for learning, but it still seems to be popular. Is it a philosophy of "no pain, no gain"? Hmm, perhaps "no pain, no pain" is better. The nitty-gritty internal workings can be learnt more easily when you are comfortable with the usage – you don't have to be able to design a string class before you can use one. This is, after all, the whole point of encapsulation.

For the same reason, templated container classes should be introduced early on. Collections of data are so common in programs that they can hardly be considered advanced, and therefore something to save until last. It is far easier to explain that `vector<int>` can be read as "vector of int", where `vector` is a resizable array type, than all the details behind `int *`, the relationship between arrays and pointers, and how to use `new[]` and `delete[]` correctly. Similarly, explaining that `std::` means simply "from the standard library". It does not require a long and detailed explanation of C++'s scope model and issues arising from programming in the large.

## The Standard Library

- The standard library provides many utilities that programmers commonly need
  - ♦ I/O, strings, resizable arrays, linked lists, data manipulation operations, numeric support, etc.
- Most of these utilities are templated so they can work with any new user-defined type
  - ♦ Programmers discouraged from reinventing the wheel... again



The standard library, and in particular the STL (as its name suggests), makes extensive use of templates to provide a variety of utility types and functions. Abstract data types and algorithms form the main body of the templated features in the library, and programmers are encouraged to look in the library to see what features are available either to use directly or that may be used in building their own specific types.



## I/O

- In addition to the C I/O library, C++ has its own I/O streams library
  - ♦ Operator overloading gives it a shell-like syntax

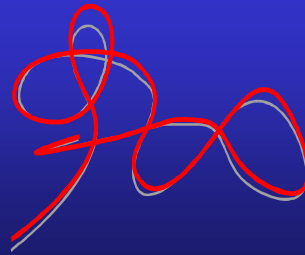
```
int main(int argc, char * argv[])
{
    std::cout << "spacer? ";
    std::string spacer;
    std::cin >> spacer;
    std::string args;
    for(size_t arg = 1; arg != argc; ++arg)
        args += argv[arg] + spacer;
    std::cout << args << std::endl;
    ...
}
```

The C++ I/O streams library adopts a shell-like notation for its input and output notation. Operator overloading has been used so that stream insertion is shown using << and stream extraction using >>. Whilst it may be argued that this was originally an abuse of the bitshift operators, it is now considered to be an ingrained C++ idiom.

The `std::cout` object corresponds to C's `stdout`, `std::cin` corresponds to `stdin` and `std::cerr` corresponds to `stderr`. Multiple values may be streamed into or from these stream objects, so that << and >> are said to be chainable. `std::endl` is conventionally used to print a newline and flush the output, which is otherwise buffered until either an explicit flush or a demand for input on `std::cin`.

## *std::string*

- A standard library type for strings supports better abstraction and safer use than *char \**
  - ♦ Supports implicit conversion from *char \**
  - ♦ Supports embedded null, *\0*, characters
  - ♦ Supports concatenation as addition
- Encapsulates and automates all of its memory management
  - ♦ Allocation and deallocation are managed internally, not by the programmer



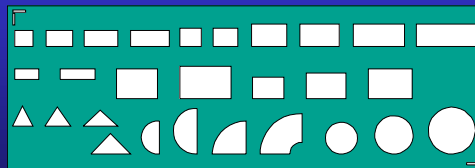
The standard library string type is far superior to vanilla `char` arrays for representing string data. `std::string` supports overloaded operators to make operations such as concatenation and comparison more intuitive — using `+` and `+=`, and `==`, `!=`, etc. rather than named functions. It also supports implicit conversions from `char *`, which means that in many places that a `std::string` is expected, a `const char *` will also serve. For programmers that need to work with binary rather than text strings that include embedded null characters, `std::string` has no problems.

From the C programmer's perspective, the most noticeable convenience of using `std::string` is the memory management, or rather the lack of it. `std::string` is properly self-contained so that the programmer does not have to manually allocate, reallocate or deallocate the memory used for the internal representation of the string. Resizing strings is a simpler and safer affair.

To use the `std::string` type you must include the standard `<string>` header. It also has a wide-character counterpart, `std::wstring`, also available in `<string>`. It is based on `wchar_t` rather than `char` elements. Some platforms map `wchar_t` to 16-bit or 32-bit Unicode.

# The Standard Template Library

- The STL originated in the work of Alex Stepanov and others
  - ♦ Both revolutionary in design and native to C++
- Incorporated into the first C++ standard at a late stage in the standardisation process
  - ♦ Displaced previous, more simplistic utilities
  - ♦ Changes made to fit it into the standard



The STL makes heavy use of templates and no use of runtime polymorphism. The library's philosophy is to make algorithm use and design significantly easier. The non-inheritance approach comes as a surprise to many, but this should not be taken to mean that the library components are inflexible. On the contrary, components are heavily parameterised, the difference being that most of the parameterisation is at compile time. It is easy to build an efficient polymorphic container hierarchy using STL containers as the underlying implementation, but not vice-versa. As such, the STL constitutes the more fundamental approach to library components.

It was the relatively late adoption of the STL in the standardisation process that held up the standard. The previous — now simplistic — utilities were jettisoned and the STL integrated. Some existing types, such as strings, were modified to fit in with the STL. In other places, the language was modified to better serve the STL, which led to further changes across the library.

## *std::vector*

- A standard library type for resizable arrays
  - ♦ *std::vector* variables automatically manage their own memory and track their own size
- Must be declared with the appropriate element type specified in angle brackets

Can be read aloud as "standard vector of *int*"

```
std::vector<int>          ints;  
std::vector<std::string>  strings;  
std::vector<std::vector<int> >  vectors;
```

The space is significant in C++98, but in C++11 onwards — and many C++98 compilers — it is tolerated

The `std::vector` type is in many ways superior to the normal built-in arrays for representing array data. The `std::vector` type should be thought of strictly as a user-defined type that behaves like a resizable array, and should not to be confused with a mathematical vector.

As with `std::string`, from the C programmer's perspective, the most noticeable convenience of using `std::vector` is the automated memory management. `std::vector` is properly self-contained so that the programmer does not have to manually allocate, reallocate or deallocate the memory used for the internal representation of the array.

To use the `std::vector` type you must include the standard `<vector>` header.

## Beyond the Standard Library

- The free Boost libraries complement the facilities available in the C++98 library
  - ♦ Open source community offering proven and reviewed code for a range of tasks
- TR1 (Technical Report 1) defines an ISO supplement to the standard library
  - ♦ Drawn extensively from Boost
- Much of TR1 has been incorporated into the C++11 standard
  - ♦ As well as many other language mechanisms

More properly TR1 is known as the *Draft Technical Report on C++ Library Extensions*. The intention behind it was as preparation for C++11, the next standard. This approach ensures that prior art backs up future library expansion, especially as it is based on much mature work in the Boost libraries. Following the current C++ standard (C++14) work in progress on the other libraries will be completed and released as separate numbered Technical Specifications (TSs).

The TR1 library is divided up in terms of general utilities, function objects, metaprogramming and type traits, numerical facilities, containers, regular expressions and C99 compatibility. More specifically it provides the following key features for the C++ programmer:

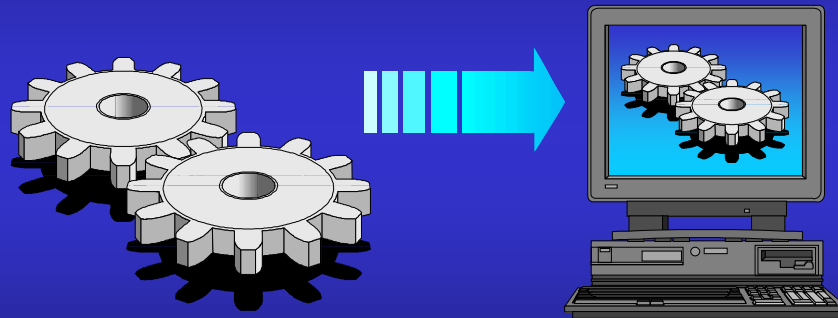
- Memory-managing smart pointers: `shared_ptr` and `weak_ptr`.
- Containers: unordered associative containers (i.e. hashed containers), `array` and `tuple`.
- Function objects: `function` and `bind`.
- Regular expressions.
- Random number generation.

All TR1 library extensions are defined in namespace `std::tr1`.

## Summary

- C++ is a language with support for multiple programming styles
  - ◆ Including procedural, object oriented and generic
- Defining new classes forms the basis of much C++ development
  - ◆ Derivation, *virtual* function overriding and operator overloading support composition and variation
- The C++ standard library includes a number of commodity features for common use
  - ◆ I/O, containers, exceptions, etc

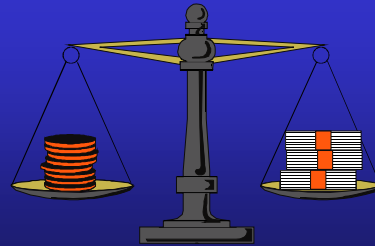
# Conversions and Mutability



*C++ Advanced*

# Conversions and Mutability

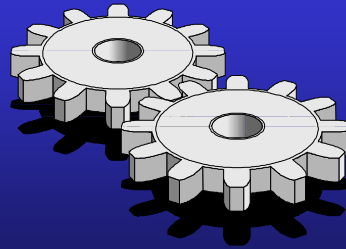
- Objectives
  - ♦ Introduce and restrict type conversions
- Contents
  - ♦ Keyword casts and RTTI
  - ♦ Converting constructors and *explicit*
  - ♦ User-defined conversion operators
  - ♦ *const* qualification
  - ♦ Overloading on *const*
  - ♦ *mutable*





# Conversions

- A conversion may be implicit or explicit
  - ♦ *Implicit* places control with the compiler
  - ♦ *Explicit* places control with the developer
- Conversions may be widening or narrowing
  - ♦ A widening conversion is always safe, i.e., a generalisation that does not lose precision
  - ♦ A narrowing conversion is not and should be explicit, i.e., may lose precision



Conversions may be implicit or explicit, which places them under the control of the compiler and the developer, respectively. Whether a conversion should be implicit or explicit is a mixed matter of taste, safety and requirement. Widening conversions, where a conversion is from a specific to a general type, are always safe and can be implicit without offending sensibilities or the compiler, e.g., `int` to `double` or derived to base. Narrowing conversions, where a conversion is from a general to a specific type, are not guaranteed to be safe and should be explicit. One would hope that narrowing conversions would be required to be explicit, but this is not always the case, e.g., `double` to `int`. Even though the compiler does not require it, one might argue that taste does. Where possible, narrowing conversions should be checked, e.g., the use of `dynamic_cast` to go from base to derived.

User defined conversions, through converting constructor and user defined conversion operators, can allow a developer control over class miscibility. This is useful for value-based classes and where Type Heterarchies need defining.

Copying is a degenerate form of conversion, and therefore includes copy constructors and assignment operators. Other assignment operators are also considered to express a form of conversion, i.e., the ability to use an object on the right hand side that is of a different type to the left hand side. Conceptually the conversion is expressed in the implementation of the assignment operator.

Conversions can also be constrained through Preventative Overloading, where member functions are defined as private to intercept unwanted implicit conversions. This is common in preventing copying for non-value-based classes, but can also be used to good effect in other classes for other conversions.

## Keyword Casts

- C++ offers a safe and clear family of keyword casts, in addition to the C and function styles
  - ♦ Each cast performs only one type of conversion

**static\_cast**

*Used for numeric and common conversions, such as void \* to another pointer type*

**dynamic\_cast**

*Part of C++'s support for object orientation, used for converting between pointers to related types*

**const\_cast**

*Used to loosen the const or volatile qualification of pointers*

**reinterpret\_cast**

*Used for changing between unrelated types, e.g. between int and pointer or unrelated pointer types*

The four keyword casts address issues of clarity and reliability. Each one is a specialist, good at only one type of conversion. Based as they are on keywords and a very explicit syntax, they are very easy to locate in your source code.

**static\_cast:** This cast covers all of the safe casts and some of the moderately unsafe ones. For instance, conversions between different numeric types and `void *` to typed pointers.

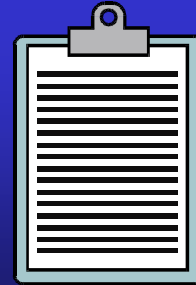
**dynamic\_cast:** This cast handles type-safe casting between classes related by inheritance that support polymorphism.

**const\_cast:** This cast handles changes in cv qualification, i.e. `const` and `volatile` qualification, for pointers. The most common use of this form is to remove `const`-ness, hence its name, and in particular from poorly written code that is not `const` correct. None of the other casts permit changes in qualification, and will flag any attempts as compile time errors.

**reinterpret\_cast:** This cast plays the role of the low-level cast beloved of systems programmers: unportable and dubious casts such as converting between pointers and integers, pointers to data and pointers to functions, etc. Here be dragons with attitude.

## Runtime Type Information (RTTI)

- Classes with *virtual* functions support dynamic runtime type queries
  - ♦ Safe downcasting using *dynamic\_cast*
  - ♦ Additional type information, such as the name of an object's actual class, through *typeid*
- These operations reveal an object's actual type, not the declared pointer or reference type



There are times when a piece of code must know the actual type of the object being pointed to with a base class pointer. Rather than letting the programmer loose with arbitrary casts, C++ provides the RTTI mechanism.

To confirm that an object is a specific type, the `typeid` operator can be used:

```
window * child = ...;
if(typeid(*child) == typeid(dialog))
    ...
```

The standard `<typeinfo>` header must be used to allow this code to be compiled. If a recast pointer to the object is what is needed, the `dynamic_cast` operator should be used:

```
window * child = ...;
if(dialog * child_dialog = dynamic_cast<dialog *>(child))
    ...
```

It returns null for pointer casts that fail, i.e., point to a different type of object, and throws an exception if the cast fails for a reference type.

## Converting Constructors

- A single argument constructor also provides a conversion from the argument type to the class
  - ♦ Implicit conversions occur when a value or a *const* reference is expected
  - ♦ Does not apply for non-*const* references

```
void lookup(const url &);  
url home = "http://curbralan.com";  
lookup("http://asemantic.net");  
url mail("mailto:info@curbralan.com");  
mail = "mailto:webmaster@asemantic.net";
```

*What would the constructor look like for the url class?*

The following definition would satisfy the code above:

```
class url  
{  
public:  
    url(const char *);  
    ...  
};
```

An conversion can be supported by the introduction of one or more converting constructors on a class. These allow conversions into a type from other types.

## *explicit*

- Declaring a constructor *explicit* prevents it being used as a conversion
  - ♦ It is not appropriate for all single argument constructors to act also as implicit conversions

```
class file
{
public:
    explicit file(const std::string & name);
    ...
};
```

```
file blog = "blog.txt"; // won't compile
file log("log.txt");    // compiles OK
```

Converting constructors should be used in support of conceptually similar types, e.g., `string` and `const char *` are both representations of strings. The more normal case is that two types are not so related, e.g., a file and its name, and an `explicit` modifier should be used on any single-argument constructor.

## Explicit Temporaries

- A temporary object can be created using syntax that looks like a direct constructor call
  - ♦ Allows temporary objects to be created where conversion is inappropriate or not possible

```
class rectangle
{
public:
    void resize(const point & corner, const offset & extent);
    ...
};
```

```
rectangle boundary;
...
boundary.resize(point(-128, 128), offset(256, -256));
```

It is not possible to define a literal form for a new type, but the constructor expression syntax comes close, e.g., `string("theory")`. This is stylistically preferable to using `static_cast` in this context as it is a well defined conversion (as opposed to a potentially dangerous conversion that must be highlighted in the source code) and corresponds well to the idea of constructing a new value. Thus programming guidelines that recommend `static_cast` instead of function (or traditional) cast notation for all such conversions are misguided, and give the code the wrong meaning, i.e., "look here, there's a suspicious narrowing conversion going on". The preferred style also means that code appears consistent when used with other multiple argument constructed forms, e.g., `string(5, '*')`.

## Implicit Temporaries

- An implicit temporary object is created when...
  - ♦ Values are passed or returned by copy
  - ♦ A value of one type is used where another type is expected, and an implicit conversion is available
- The lifetime of temporaries is until the end of the enclosing full expression

```
lookup ("http://curbralan.com") ;
```

The compiler translates to...

```
lookup (url ("http://curbralan.com")) ;
```

Temporary objects are sometimes created when working with value-based objects. For instance, the return values of operators, such as `operator+`, or the result of an implicit conversion via a converting constructor. Such temporary objects last until the end of the enclosing full expression, which is often the statement or control condition in which they were created.

## User-Defined Conversions (UDCs)

- It is possible to define conversions from instances of a class to values of another type
  - ♦ Effectively the opposite of a converting constructor

UDCs are operators whose return type is also their 'name'

I/O stream objects support usage as Booleans to determine whether they are still valid

```
class convertible
{
public:
    operator bool() const;
    ...
};
```

```
std::string word;
while(std::cin >> word)
    std::cout << word << std::endl;
```

An outward conversion can be offered by the introduction of a user-defined conversion operator. However, good uses for outward conversions are rare, and uses for multiple outward conversions in a class create more ambiguity than expressiveness – the kind of ambiguity that provides the class user with apparent gibberish for a compilation error. In the case of a `string` class, it is advised that outward conversions to a `const char*` are not supported: they can lead to dangling pointer errors.



## User-Defined Conversion Guidelines

- Conversions make sense only for value objects
  - ♦ So implicit conversions should not be provided for other kinds of objects
- Conversions make sense only if the source and target in some way represent the same concept
  - ♦ Otherwise make converting constructors *explicit*
- UDC operators should in general be avoided
  - ♦ Can cause too much user confusion and compile-time ambiguity

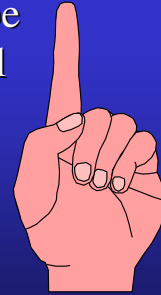


Conversions may be implicit or explicit, which places them under the control of the compiler and the developer, respectively. Whether a conversion should be implicit or explicit is a mixed matter of taste, safety and requirement. Widening conversions, where a conversion is from a specific to a general type, are always safe and can be implicit without offending sensibilities or the compiler, e.g., `int` to `double` or derived to base. Narrowing conversions, where a conversion is from a general to a specific type, are not guaranteed to be safe and should be explicit. One would hope that narrowing conversions would be required to be explicit, but this is not always the case, e.g., `double` to `int`. Even though the compiler does not require it, one might argue that taste does. Where possible, narrowing conversions should be checked.

User-defined conversions, such as through converting constructor, can allow a developer control over class miscibility. This is useful for value-based classes.

## Qualification and Clarification

- In C++ *const* qualification plays a role in interface design and encapsulation
  - ♦ Functions in a class interface that are *const* qualified are clearly intended as query operations
- *const* qualification of functions should be based on logical *const*-ness, not physical
  - ♦ The default meaning of *const* implies physical *const*-ness



In C++ passing a value object around by `const` reference offers immutability through a level of indirection that is mostly transparent. This is in contrast to using pointers for that level of indirection, which require a change in usage notation unsuitable for values but appropriate for other object types.

## *const* Qualification and Interfaces

- From the perspective of qualification every class has multiple interfaces
  - ♦ For instance, with respect to *const* a class can be considered to have a *const* interface and a non-*const* interface
- The qualified interface may be considered a supertype of the unqualified one
  - ♦ I.e., the unqualified one is more specialised with respect to mutability than the qualified one and can be used where the qualified one is expected

Explicit separation of modifier from query functions can benefit a system, and this is a concept that can be expressed in C++ using type qualifiers. Thus `volatile` and `const` – as well as `mutable` – are unified under the heading of change, even if the names are not as well chosen as they might be (that access to an object may be `const` `volatile` is a curio that leaves many developers bemused). However, it is normally `const` that is used and discussed.

We can see that substitutability plays a part with respect to `const` when we consider that any C++ class typically has, in effect, two public interfaces: the interface for `const` qualified objects and the interface for non-`const` qualified objects. The `const` interface is a subset of the non-`const` interface, and therefore a non-`const` object may be used wherever a `const` one is expected. Note that the subtyping relationship implied need not be strict, as overloading can be used to block functions from default access, i.e. overloaded functions differing only in respect of `const` or non-`const` will be selected according to the calling `const`-ness.

## Overloading on *const*

- Can overload functions with respect to *const*-ness of references or pointers
  - ♦ Useful where the result of a function should reflect the *const*-ness of its argument

```
char & find(char * text, char to_find);  
const char & find(const char * text, char to_find);
```

```
const char & find(const char * text, char to_find)  
{  
    while(*text && *text != to_find)  
        ++text;  
    return *text;  
}
```

Because a `const` pointer or a `const` reference is considered distinguishable from the corresponding non-`const` version, it is possible to overload with respect to the `const`-ness of pointers or references. This is useful when an algorithm needs to be provided in two different flavours: one that works on `const` data and returns only `const` access to it, and another that works on non-`const` data and returns non-`const` access to it.

In the standard C library there are cases where `const`-ness is broken. The `strchr` function returns a pointer to the first occurrence of a matched character in a string:

```
char * strchr(const char *, int);
```

Notice that the argument is in terms of `const` but the result, which points into the same string as the argument, returns non-`const`. This situation is fixed in C++:

```
namespace std  
{  
    const char * strchr(const char *, int);  
    char * strchr(char *, int);  
}
```

## Subscripting with *operator[]*

- The subscript operator is useful for containers or container-like classes
  - ♦ Must be defined as a member function

```
class polygon
{
public:
    ...
    const point & operator[](std::size_t index) const
    {
        return points[index];
    }
    ...
private:
    std::vector<point> points;
};
```

The subscript operator can be overloaded as a member function. It can take any argument type as an index or key, not just integer types. However, the question arises concerning its return type: although a query function, it is normally possible to assign through the result of `operator[]` on non-const objects. How is such an operator defined?

## Overloading Subscripting on *const*

- Support for intuitive subscripting semantics
  - ♦ Result has same *const*-ness as container

```
class polygon
{
public:
    ...
    const point & operator[] (std::size_t index) const;
    point & operator[] (std::size_t index);
    ...
private:
    std::vector<point> points;
};
```

```
polygon shape;
...
const polygon fixed_shape = shape;
std::cout << fixed_shape[0]; // compiles OK
fixed_shape[0] = point();    // won't compile
```

The technique that resolves this tension is to overload the subscript operator with respect to *const*-ness. The *const* variant is chosen by the compiler for *const* objects and the non-*const* variant for non-*const* objects. The return type follows the appropriate qualification, granting no more or no less access than is appropriate for the qualification of the container type.

## *mutable*

- Allows *const* functions to modify member data
  - ◆ Bridges physical and logical *const*-ness

```
class dictionary
{
public:
    const std::string & operator[](const std::string & key) const
    {
        if(key != cached_key)
        {
            cached_value = ...; // lookup value by key
            cached_key = key;
        }
        return cached_value;
    }
    ...
private:
    mutable std::string cached_key, cached_value;
    ...
};
```

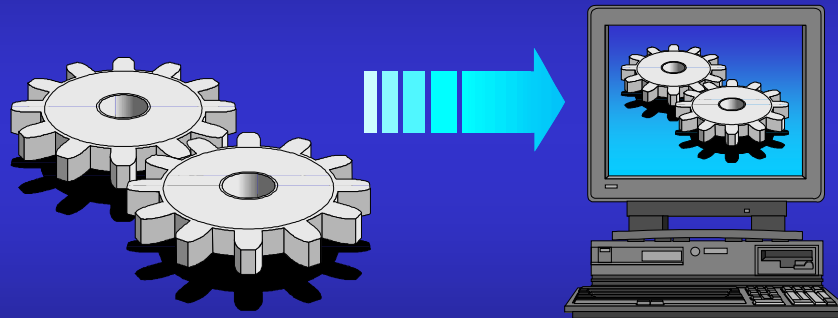
Interfaces should present a logical view of an object's behaviour, and hence `const` should reflect logical queries. `mutable` can be considered a feature of negative variability that allows expression of physical change behind a logically immutable façade. It allows separation of logical and physical `const`-ness, for instance in the case of caching.

## Summary

- Conversions may be provided for user-defined types, i.e., classes
  - ♦ Converting constructors and UDCs
  - ♦ *explicit* can be used to prevent unwanted conversions for single-argument constructors
- The *const* qualifier represents both an interface and an implementation design feature
  - ♦ It is possible to overload with respect to *const*-ness
  - ♦ *mutable* decouples logical interface *const*-ness from physical implementation *const*-ness



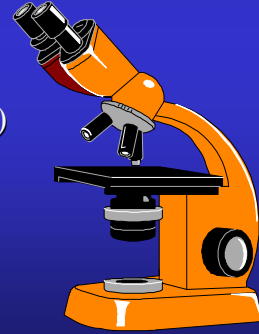
# Scope



*C++ Advanced*

# Scope

- Objectives
  - ♦ Introduce the scope-management features of C++
- Contents
  - ♦ Namespaces
  - ♦ *namespace std*
  - ♦ Argument dependent lookup (ADL)
  - ♦ Class nested types
  - ♦ Class *statics*



# Namespaces

- Namespaces define a named scope for declarations and definitions
  - ♦ Helps to avoid name clashes
  - ♦ Namespaces may be reopened, so that they can span many header files

```
namespace file
{
    struct handle {...};
    handle * open(const char * name);
    void close(handle *);
    std::size_t read(handle *, char * to, std::size_t max);
    std::size_t write(handle *, const char * from);
    ...
}
```

file.hpp

Namespaces can be used to name subsystems or just small collections of features in a header. In terms of a modular approach to programming, they allow programmers a more structured naming scheme for elements within their program than just a flat, global namespace.

Modules are normally decomposed according to design decisions, i.e. all those features that will be affected together, or arise together, from a common design decision are grouped together in a module. This often leads to modules being organised around related services or functions on a data type, as shown above.

Syntactically, namespaces do not have a trailing semi-colon. They can also be nested, although such decomposition is used less often than in other languages with equivalent scoping constructs.

## Accessing Names in Namespaces

- Can access names in a namespace...
  - ♦ Explicitly using the fully scope qualified name
  - ♦ Via a *using* declaration that pulls in specific names
  - ♦ Via a *using* directive that pulls in all names

```
#include "file.hpp"
...
using file::open;
static file::handle * log = open("log.txt");
void write_log(const char * message)
{
    using namespace file;
    write(log, message);
}
```

To access an element from within a namespace, the fully scoped name can be used, e.g., `file::handle`. The contents of a whole namespace can also be imported with a *using* directive, e.g., `using namespace file;`. However, this can sometimes defeat the visibility and traceability of program organisation that is offered by namespaces. Individual features can be imported with *using* declarations, e.g., `using file::handle;`. Any clashes that emerge as a result of *using* can be resolved with explicit qualification.

Also note that namespace aliases allow programmers to give alternative names, often abbreviations, of a namespace, e.g.,

```
namespace abbr = long_namespace_name;
```

# Global and Anonymous Namespaces

- The global namespace has no name
  - ♦ The scope resolution operator and no scope name accesses names in the global namespace
- Anonymous namespaces keep definitions private to a source file
  - ♦ A more general mechanism than file-scoped *statics*

```
#include "file.hpp"
...
namespace
{
    file::handle * log = file::open("log.txt");
}
...
```

Identifiers at the global or anonymous namespace level can be using a simple, unary-like scope resolution, e.g.,

```
::FILE *log = ::fopen("log.txt", "a+");
```

Note that although it is unlikely they will ever be removed, file-scope `static` variables have been deprecated in favour of the more regular unnamed namespace concept.

## *namespace std*

- The C++ standard library is in namespace *std*
  - ♦ The C++ standard library does not use *.h* or *.hpp* suffixes on its headers, e.g. `<iostream>`
- The C library is part of the C++ library
  - ♦ Nested in namespace *std* if new header style is used, e.g. `<stdlib.h>` versus `<cstdlib>`

```
#include <iostream>
int main()
{
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

The standard C library is incorporated into the standard C++ library. Features are available in their traditional form through the traditional headers, e.g. `<stddef.h>`, or wrapped in namespace `std` from headers in the new style, e.g., `<cstddef>`.

To see the difference, consider a more traditional C implementation of the code above:

```
#include <stdio.h>
int main()
{
    puts("Hello, World!");
    return 0;
}
```

This is perfectly fine as C++, but to make the origin of components clearer, here is a more explicit C++ version:

```
#include <cstdio>
int main()
{
    std::puts("Hello, World!");
    return 0;
}
```

## Argument-Dependent Name Lookup

- It is common and recommended practice to define types in a namespace
  - ♦ Reduces risk of collision and can clarify role
- Any associated non-member functions will also be in the same namespace
  - ♦ Lookup searches for matching functions in the namespaces of its argument types
  - ♦ However, this can be subtle and mysterious when errors arise



To access an element from within a namespace, the fully scoped name can be used, e.g., `std::string`. The contents of a whole namespace can also be imported with a `using` directive, e.g., `using namespace std;`. However, this can sometimes defeat the visibility and traceability of program organisation that is offered by namespaces. Individual features can be imported with `using` declarations, e.g., `using std::string;`. Any clashes that emerge as a result of `using` can be resolved with explicit qualification.

For functions, especially operators, an alternative that requires no explicit import is to rely on argument-dependent name lookup (also known as Koenig lookup): an unqualified function will be first looked up in the namespaces of its argument types. This means that using the non-member stream insertion operator, `operator<<` defined in `std`, with `std::cout` and a `std::string` will work without having to import the operator with a `using` declaration. Note that some compilers still do not implement full ADL.

It is generally best to avoid relying on ADL more than is strictly necessary. The feature can undermine the modular structure of a program and is both subtle and awkward when unexpected names are pulled in.

## Nested Types

- Other types may be nested within a class
  - ♦ Type names accessed via scope resolution operator

```
class file
{
public:
    enum mode
    {
        none, reading, writing, any
    };
    ...
};

file::mode open_mode = file::writing;
log.open("back.log", open_mode);
```

A class is not simply an abstraction for objects, describing their type. It can also play the role of a named scope, like a mini-namespace. This is useful for describing concepts that are tightly coupled and very closely allied to the object type itself. The scoping ensures that the type name does not clash with any similarly named concepts used elsewhere in a program.

For instance, the code above shows a `mode` type for use with a `file`. The file can be opened using the appropriate mode, and the mode can also be queried:

```
class file
{
public:
    enum mode
    {
        none, reading, writing, any
    };
    bool open(const std::string &, mode = any);
    mode opened_as() const;
    ...
};
```

`file::none` is returned if the file object is closed.



## *static* Member Functions

- *static* member functions are called with respect to the class scope rather than a target object
  - ♦ They have no *this* pointer

```
class file
{
public:
    static file * open_for_reading(const std::string &);
    ...
};
```

```
file * file::open_for_reading(const std::string & name)
{
    file * result = new file;
    result->stream = fopen(name.c_str(), "r");
    return result;
}
```

Note that the `static` keyword is not permitted on the actual definition of the member function.

It is possible to call a `static` member function with respect to an object as well as by its scope. However, this is just a syntactic convenience: nothing actually happens to the target object.

## *static* Member Data

- *static* data can also be declared within a class
  - ◆ Requires a separate definition in addition to the declaration in the class body
  - ◆ *static* data often hints at a missing design concept

```
class file
{
    ...
private:
    static file * last_opened;
    ...
};
```

```
file * file::last_opened; // initialises to null
```

Along with `static` functions, which are particularly useful for traits and factory functions, it is also possible to define `static` class data. In essence, these are just globals with scope etiquette, and will suffer and cause the same problems as global variables normally do in a design — so use sparingly for non-`const` data.

## *static const* Member Data

- The safest and most useful *static* data is *const*
  - ♦ For integral types initialised by a compile-time constant initialisation in the class body is possible
  - ♦ Otherwise initialisation and declaration are separate

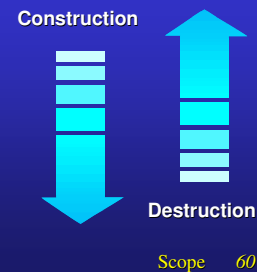
```
class file
{
public:
    static const char separator = '/';
    static const std::string protocol;
    ...
};

const char file::separator;
const std::string file::protocol = "file:";
```

Note that the special dispensation for initialisation in the class body extends only to integral types such as integers and `enums`: floating-point numbers and user-defined types are not accorded the same privilege. However, an uninitialised definition is still required because the constant needs a unique location in memory, which will be associated with the definition.

## *static* Initialisation Order

- Within a translation unit, *static* data is initialised in order of definition
  - ♦ Applies to all data defined at file level, not just data declared *static*
  - ♦ The order between translation units is not specified
- Within a function, *static* data is initialised on first call
- Destructors are *always* executed in reverse order of construction



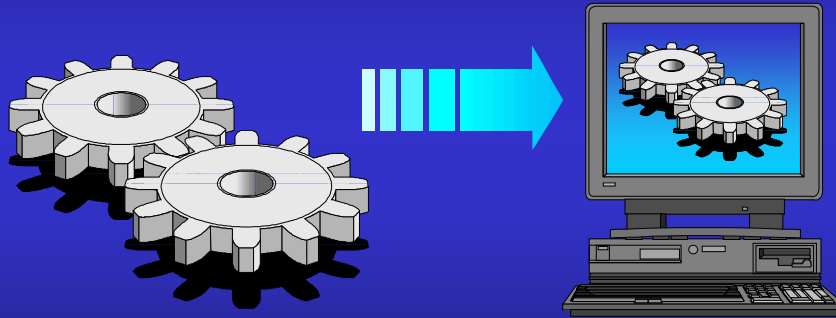
© Curbralan Ltd

Because of the deferred initialisation of `static`s and the unspecified total ordering between translation units (as typically represented by `.cpp` files), programmers often resort to a number of tricks to ensure some kind of ordering. Perhaps the simplest advice is to avoid modifiable `static` data of any sort and focus on designing clearer object relationships instead of resorting to programming tricks.

## Summary

- Namespaces allow libraries and subsystems to be grouped and labelled consistently
  - ♦ Namespaces can span multiple files
  - ♦ Unnamed namespaces can be used to hide details within a source file
- Classes also define scopes that can be used for non-object purposes
  - ♦ Type declarations, type definitions, *static* functions and *static* data can all be nested within a class

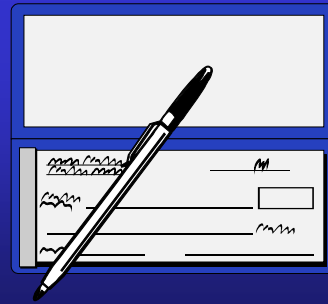
# Value-Based Objects



*C++ Advanced*

# Value-Based Objects

- Objectives
  - ◆ Outline the idioms appropriate for working with value objects
- Contents
  - ◆ Anatomy of a value object
  - ◆ Operator overloading
  - ◆ Construction and destruction
  - ◆ Copying and prevention
  - ◆ Comparison



# Value Objects

- Values are referentially transparent objects
  - ♦ Typically not in a class hierarchy
  - ♦ Strongly informational but no separate identity
  - ♦ Not intended to be shared between objects
- Defining a class for value objects...
  - ♦ Support simple construction and copying
  - ♦ Uniform use suggests pass by copy or *const* reference, but not pointer
  - ♦ May support limited conversions



Values are objects for which identity is not significant, i.e., the focus is principally on their state and then their behaviour. This is reminiscent of data types and instance in functional programming languages. Other distinguishing features of values include their granularity and content: they are typically fine-grained rather than coarse, with behaviour closely structured around their state.

In C++, values are associated with a set of capabilities and conventions. A string is an example of a value object: the focus is on its content and its manipulation but not on its address in memory, i.e., comparison of the content of two strings is of interest, but comparison of their identity is less useful. One value is substitutable for another with the same state. Thus in C++, value-based programming relies heavily on `const`, aliasing through references, the ability to copy by construction and assignment, and typically little or no involvement with class hierarchies. However, simply because a value can be copied, it does not mean that it must be copied — this is part of the transparency. This is in contrast for objects whose identity is significant, where copying is often not meaningful — and is therefore disabled — and which often play a role in a class hierarchy.

A common question asked by many C++ developers is when to pass objects around by pointer and when to pass by reference. This is perhaps the wrong question, but it can be resolved by an appeal to common usage: Values are typically stack objects or data members, and they will often support operator overloading. This suggests that they should always appear either as references or directly declared objects. Objects for which identity is significant should be passed around with their identity significant, i.e., by pointer.



## Anatomy of a Value Object

- Tend to have common interface features
  - ♦ These features are common rather than mandated

```
class value_type
{
public:
    value_type();
    value_type(const value_type &);
    value_type(const other_value_type &);
    ~value_type();
    value_type &operator=(const value_type &);
    ...
};

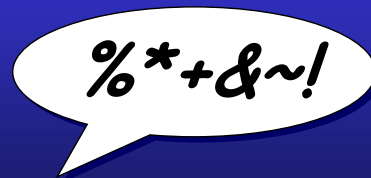
bool operator==(const value_type &, const value_type &);
bool operator!=(const value_type &, const value_type &);
... // relational operators iff is an ordered quantity
```

The features shown for a typical value object type are just that: typical. They are not mandatory, but their absence might sometimes be surprising. For example, it is highly suspicious if a value type does not support assignment or copy construction. Less surprising if it does not support default construction, although this is expected. And less surprising in turn if equality and inequality are not supported. Relational operators are only expected for value objects that represent ordered quantities, so their absence is inappropriate in fewer cases — indeed their presence may also be inappropriate, e.g., trying to invent a meaning for types with no natural ordering.

Note that this class should be considered a pseudo-class: the expectations also apply to built-in types and classes that default the special member functions.

# Operator Overloading

- Operators can be intuitive for value objects
  - ♦ But make little sense for heap objects
- Common name implies common purpose
  - ♦ Operators provide a predefined set of names and definitions, and therefore expectations
- Operators should be considered part of the class interface
  - ♦ Regardless of whether or not they are actually class members



Overloading is based on the idea that common name implies common purpose, and this frees programmers from indulging in their own name mangling to differentiate similar functions (this is the job of the compiler). Overloading works closely with — and sometimes against — conversions. Developers are cautioned to keep any eye open for any such interference.

Extension through overloading forms an important part of operator overloading. For instance, a class that is closed to change can apparently be extended to work within a framework, e.g., 'extending' I/O streams to understand new types. Some extension is less transparent, but importantly follows as much of the base form as possible. An obvious example is the use of placement new operators which, in spite of taking additional arguments, have the same purpose and much of the same form as the vanilla new. Tagged overloads, such as `new(std::nothrow)`, provide a means of compile time selection that is a compromise between operators and named functions.

A combination of language constraints and common practice idioms suggest which operators should be members and which should be global. Regardless of this, operators should be considered a part of the class interface.

## Operators Follow Built-ins

- What guidelines should the behaviour of overloaded operators follow?
  - ♦ Compilers neither care nor check
- Built-in and library operators set expectations and offer a familiar set of behaviours
  - ♦ Follow their lead where possible

***When in doubt, do as the `ints` do.***

**Scott Meyers**

The founding good practice for operator overloading can be considered to be that operators should follow built-ins with respect to signature and behaviour. Such set of recommendations is common and can be found in many places, and is further formalised in generic programming requirements. For any user of the code this idiom supports the principle of least astonishment. Well-design smart pointers can already be said to have examples of this practice.

Where the implied semantics of built-ins cannot be met, other accepted idioms, such as streaming, provide a second port of call. Thus we can temper a hard line view of operator overloading with a little – but not too much – pragmatism:

*"In view of this philosophy of only overloading operators intuitively, how do you explain the first example in your book, which shows an overloaded Left Shift << operator acting as a stream operator?"*

"Good question. We have a saying in Denmark: 'Don't do as the priest does, do what the priest says.'"

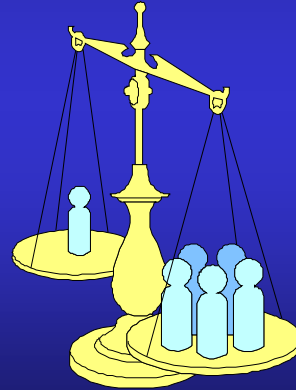
EXE interview with Bjarne Stroustrup, 1990

For some cases, overloading can be considered unreasonable: because the short circuiting evaluation cannot be emulated for `&&`, `||` and `,`, they should not be overloaded. For others, a little laxness in interpretation is often considered reasonable, e.g., the use of `operator+` for string concatenation, which is clearly not commutative.

Where operators redefine something fundamental, such as memory management, writing to common form becomes more than just a courtesy: failure to follow form can corrupt a system.

## Balance Overloaded Operators

- What does 'completeness' mean for operators provided for a type?
  - ♦ Completeness is related to expectation and ease of use
- Relationships exist between built-in operators
  - ♦ Overloading one operator often leads to overloading others



© Curbralan Ltd

Value-Based Objects 68

Interfaces should be as complete as is meaningful. But what does this mean for overloaded operators? One can extend the basic advice of having operators follow built-ins to cover relationships between operators. This results in a need to balance overloaded operators. There is a set of expectations that should be met by the class developer: operator overloading comes with a greater set of obligations than the compiler will check.

Class users have the right to full functionality. For instance:

- Equality as `operator==` implies `operator!=`.
- Relational comparison in the form of `operator>` implies, in addition, `operator>=`, `operator<` and `operator<=`.
- Prefix `operator++` implies postfix `operator++`.
- Binary `operator+` implies `operator+=`.
- Dereference `operator*` implies `operator->`.

They can also reasonably expect such behaviour to be consistent. For instance:

- Equality and inequality are clearly related, such that `a != b` should be equivalent to `!(a == b)`.
- Symmetric operators overloaded to ensure symmetry.

Note that the use and style of the templated operators in `std::rel_ops` is a questionable shortcut in gaining such balance.

## Member vs Non-member Operators

- Some operators must be member functions
  - ♦ E.g., copy assignment and subscripting
- Some operators are inevitably non-members
  - ♦ E.g., for supporting I/O for user-defined types
  - ♦ When conversions are required on both operands
- Symmetric binary operators are normally non-members
- Unary operators and other binary operators are normally members



Certain operators defined for a class must be declared globally (or more accurately, at namespace scope) to support the expected symmetry of these operators. Consider the following:

```
class string
{
public:
    string operator+(const string &) const;
    string operator+(const char *) const;
    ...
};
```

This class supports catenation for two `string` objects, and for a `string` object on the left and a `const char *` on the right, but not unfortunately for a `const char *` on the left and a `string` object on the right. The same issue applies if one requires conversions on the left hand side, i.e. if the second overload above were omitted the code that compiled would still compile, and the code that did not still would not.

## Operator Consistency

- Operators that must be global do not have access to the private section of a class
  - ♦ Member operators automatically have this access
- Define global operators in terms of member functions and other operators
  - ♦ There is no need to use *friend* functions

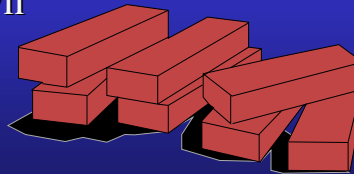
```
offset operator+(const offset & lhs, const offset & rhs)
{
    offset result = lhs;
    result += rhs;
    return result;
}
```

The temptation is to declare the global operators as `friend` functions to allow them access to the private section of the class. However, for the number of operators this creates an excessive — and unnecessary — amount of coupling.

Operators often offer syntactic sugar for functionality that either already exists in a class or can be expressed in terms of existing class primitives. This specific approach layers operators and functions, and reduces the effects of change in any basic algorithms, e.g., changing `operator+=`.

## Constructibility and Destructibility

- Many value objects are default constructible
  - ♦ Convenient for I/O, arrays and containers
- Also, all value types are expected to be publicly and safely destructible
  - ♦ Destructors are not permitted to throw exceptions in well-defined programs
- Values encapsulate their own memory management

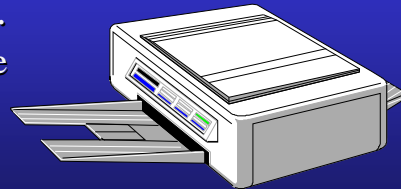


Default constructibility makes sense for value objects, even if they do not have a meaningful zero value. It allows them to be used for input and arrays. It also allows them to take advantage the full feature set of containers, such as `std::map`'s magic subscripting.

If an obvious default does not present itself, i.e. a natural 'zero', the value can represent a well-defined uninitialised state, like a NaN.

# Copying and Passing Values

- Values support copying operations
  - ♦ Copy construction and copy assignment
  - ♦ Normally arguments are passed by copy for built-ins and by *const* reference for class types
- For lightweight classes...
  - ♦ Copy operations can be default generated
- For other values classes...
  - ♦ Copy operations should be defined explicitly



Because they are fine grained and focused on information rather than identity, it makes sense to copy value objects. Copying occurs through copy construction or copy assignment. These operations are default generated by the compiler if the class author does not provide them. This makes sense for lightweight classes whose own members also have copy semantics defined, e.g.,

```
class date
{
    ...
private:
    int year, month, day;
};
```

For classes that manage their own resources, copy operations must be declared, and also a destructor defined, e.g.,

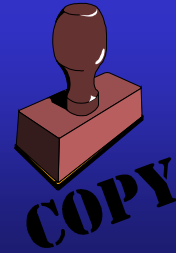
```
class string
{
public:
    string(const string &);
    string & operator=(const string &);
    ~string();
    ...
private:
    char * text; // managed using new[] and delete[]
    ...
};
```

Assignment should be self-assignment safe, i.e., if an object is assigned to itself, there should be no problems.



## Copying Semantics

- *CopyConstructible* defines role and semantics of copy constructor
  - ♦ For lightweight classes, copy operations can be default generated by the compiler, i.e., 'copyright'
  - ♦ For other values classes, copy operations *must* be defined explicitly
- *Assignable* defines role and semantics of copy assignment operator



Copying capabilities are often considered to be at the heart of what it means to be a value object. In particular, an object that is not *CopyConstructible* cannot be used in a standard container and an object that is not *Assignable* cannot be used as the target of an output iterator.

# Copy Constructors

- Copy construction can be defined for a class
  - ♦ Should be defined if managing resources that cannot or should not be memberwise copied

This is equivalent to what the compiler will attempt to do by default when initialising one *point* from another

```
class point
{
public:
    point(const point & other)
        : x(other.x), y(other.y)
    {
        ...
    }
private:
    int x, y;
};
```

A copy constructor is one that takes another instance of the same class as its only argument. By default, the compiler tries to generate a memberwise copy constructor for a class, so that each member is simply copied across directly.

# Copy Assignment

- Copy assignment can be defined for a class
  - ♦ Should be defined if managing resources that cannot or should not be memberwise copied

This is equivalent to what the compiler will attempt to use by default when assigning one *point* from another

```
class point
{
public:
    point & operator=(const point & rhs)
    {
        x = rhs.x;
        y = rhs.y;
        return *this;
    }
    ...
private:
    int x, y;
};
```

A copy assignment operator is one that takes another instance of the same class as its only argument. By default, the compiler tries to generate a memberwise copy assignment for a class, so that each member is simply copied across directly.

## Copy Prevention

- Copying should be disabled for heap objects, such as entities
  - ♦ Copying either does not make sense or becomes awkward if attempted

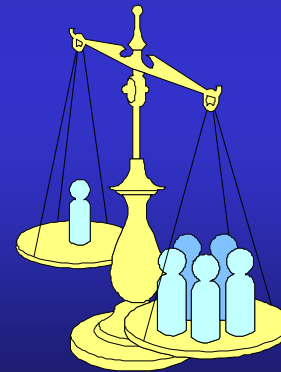
```
class account
{
    ...
    private:
    → account(const account &);
    → account & operator=(const account &);
    ...
};
```

Declared but  
not defined

Copy semantics do not make sense for entity objects or the majority of service and task objects. In these cases, copy operations should *not* be default generated as this may lead to bugs if copying is accidentally used. The class user should be prevented from using them like this accidentally, and so copy operations should be declared `private`, but no definition should be provided.

# Comparison

- *EqualityComparable* defines equality comparisons, i.e., `==`
- *LessThanComparable* defines at least a partial ordering, i.e., `<`
  - ♦ Total ordering if it defines strict weak ordering...
  - ♦ Used to determine an ordering equivalence, i.e., "not less than and not greater than" as opposed to "equal to"



*EqualityComparable* is a feature expected of many value objects. It also forms the basis of the search criteria for some of the standard algorithms. More commonly, value objects are expected to exhibit a total ordering through support for the *LessThanComparable* requirement.

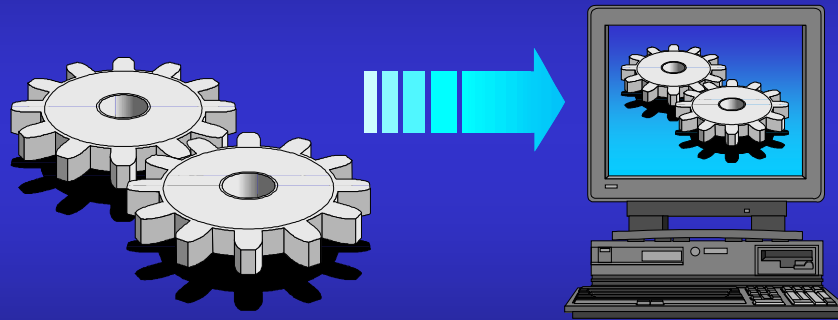
This category is not only used to order sorts and associative containers, but is also used in determining an equivalence that is subtly different to (although sometimes the same as) equality:

```
lhs == rhs                // equality
!(lhs < rhs) && !(rhs < lhs) // equivalence
```

## Summary

- Values represent simple and fine-grained information in a system
  - ◆ The focus is on their content and copyability rather than on their identity
  - ◆ Customised copy construction and assignment may be appropriate
  - ◆ Conversions and operators may be appropriate
- Non-value objects should generally not support any of these features

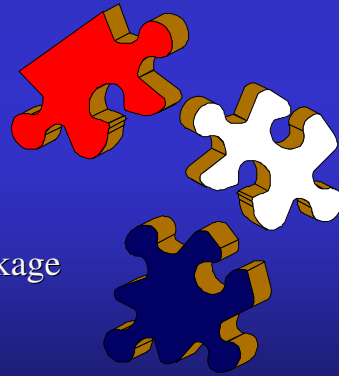
# Towards Generic Programming



*C++ Advanced*

# Towards Generic Programming

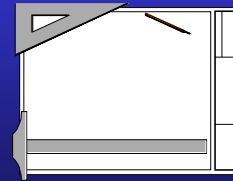
- Objectives
  - ♦ Present C++'s template mechanism
- Contents
  - ♦ Templates and generic programming
  - ♦ Function templates
  - ♦ Class templates
  - ♦ Template compilation and linkage
  - ♦ Dependent names





# Templates

- Templates play a central role in modern C++ and its associated techniques
  - ♦ Including the STL and policy-base design
- Templates factor out definition commonality
  - ♦ Applies to both function and class bodies
- Templates are properly a part of the language
  - ♦ Whereas macros are not and share none of the useful characteristics of templates



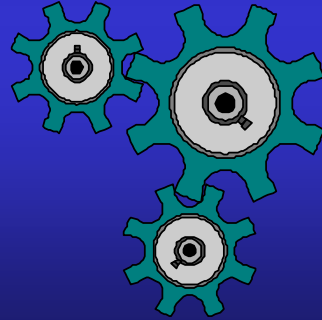
Depending on your background as a C++ developer you will either have been using templates for some time now, mostly for generic container classes, or will still be under the impression that they are a relatively recent or advanced feature of the language that you can probably live without knowing too much about.

They were originally defined in the ARM (*The Annotated C++ Reference Manual*, 1990). That they are something every C++ programmer should know about is emphasised further by looking at the C++98 standard: the most significant part of the C++ standard is the library — around 350 pages of the standard's document — and it's nearly all templated.

Even for more template-wise programmers it may come as a shock that generic, type-safe containers do constitute the main use of templates in practice. The purpose of template classes and functions is normally to generalise a data structure or algorithm with respect to a type, but template parameters may also be constants or objects with external linkage. Templates eliminate the much of the need for clumsy and error prone macros, cast obscured code, and cut and paste coding.

# Generic Programming

- An approach to program composition that emphasises algorithmic abstraction
- Built on compile-time polymorphism and value-based programming
  - ♦ Templates, overloading and conversions
  - ♦ Copying and no explicit memory management

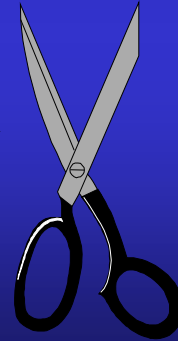


Generic programming is a form of compile time polymorphism built on C++ templates, as well as the concepts and mechanisms of the other mechanisms. The C++ standard library, notably the part known as the STL, makes extensive use of it, as do many newer designs, for instance the Boost libraries.

The use of values means that there is a reduction of memory management code, and the only level of indirection is through iterators. The emphasis on algorithmic abstraction, in combination with the use of values, gives some STL-based code the feel of functional programming.

## Refactoring Duplicate Overloads

- Overloaded functions sometimes have the same control flow structure
  - ♦ Often differing with respect to argument types
- Function templates allow common code to be written for different types
  - ♦ Types deduced from arguments at the call site and template instantiation determined at compile time
  - ♦ Ambiguities must be resolved by explicit parameter qualification or by using casts



Overloaded functions can often end up with common bodies: function templates allow this commonality to be described and factored out.

# Template Parameter Requirements

- Templates extend overloading principles
  - ♦ Template type parameters must satisfy specific, syntactic interface requirements

```
template<typename InFileType, typename OutFileType>
void file_copy(InFileType & in, OutFileType & out)
{
    std::string buffer;
    buffer.reserve(80);
    while(in.read(buffer, buffer.capacity()))
        out.write(buffer);
}
```

*What are the requirements on the template parameters in this example?*

Template parameters must satisfy requirements based on syntactic use rather than declared intention. Historically template parameters were declared using `class`, which made sense for anything that was actually a class but less so for built-in types, such as integers and pointers. A more contemporary style is to use the `typename` keyword, which more accurately reflects the intent.

# Overloading Function Templates

- Function templates can be "specialised" through overloading
  - ◆ With respect to either argument types or number

```
template<
    typename InFileType, typename OutFileType,
    typename BufferType>
void file_copy(
    InFileType & in, OutFileType & out,
    BufferType & buffer)
{
    while(in.read(buffer, buffer.capacity()))
        out.write(buffer);
}
```

As with other function forms, function templates can be overloaded with respect to argument number or type. In the latter case the expectation is that the types are clearly differentiated, e.g., one a pointer and the other a value, or they are both non-templated types.

Overloading with respect to variations also allows one overload to work in terms of another:

```
template<typename InFileType, OutFileType>
void file_copy(InFileType & in, OutFileType & out)
{
    std::string buffer;
    buffer.reserve(80);
    file_copy(in, out, buffer);
}
```

## *std::swap*

*The following is the common definition of std::swap:*

```
template<typename CopyableType>
void swap(CopyableType & lhs, CopyableType & rhs)
{
    copyable_type extra = lhs;
    lhs = rhs;
    rhs = extra;
}
```

*The following is std::swap specialised for std::vector:*

```
template<typename ValueType, typename AllocatorType>
void swap(
    vector<ValueType, AllocatorType> & lhs,
    vector<ValueType, AllocatorType> & rhs)
{
    lhs.swap(rhs);
}
```

`std::swap` is overloaded for all containers defined in the standard, normally in terms of an efficient member also named `swap`. However, the same overloading privilege is not extended to users of the library: their own containers cannot overload anything in the `std` namespace.

## *std::min and std::max*

*Comparison-based algorithms overloaded to accept comparison predicates:*

```
template<typename Comparable>
const Comparable & min(const Comparable & lhs, const Comparable & rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

```
template<typename Comparable, typename Comparison>
const Comparable & min(
    const Comparable & lhs, const Comparable & rhs,
    Comparison comparer)
{
    return comparer(lhs, rhs) ? lhs : rhs;
}
```

```
template<typename Comparable>
const Comparable & max(const comparable &, const comparable &);
template<..., typename Comparison>
const Comparable & max(..., Comparison);
```

A common example of the requirements imposed on template parameter types can be seen with `std::min` and `std::max`. The simplest forms of these function templates require their parameters to be *LessThanComparable*. The overloaded forms allow the caller to pass in their own comparison criteria.

## Member Function Templates

- Member functions of ordinary classes may also be templated individually
  - ♦ They follow similar rules to non-member templates
  - ♦ However, they cannot also be declared *virtual*

```
class file
{
public:
    template<typename StringType, typename SizeType>
        bool read(StringType & into, SizeType up_to);
    template<typename StringType>
        bool write(const StringType & from);
    ...
};
```

Just as non-member function templates decouple the type requirements for performing a general algorithm, templated member functions decouple the way in which an object can accept and work with other objects. The types need not be overly specific — just a minimal interface that satisfies the essential requirements of the function.

Where `virtual` functions allow variation of implementation but a fixed interface, member function templates support variation in interface against a fixed implementation. However, these two concepts cannot be mixed to achieve even further generalisation!



## *typename*

- *typename* plays two roles in templates
  - ♦ It can be used to introduce a type template parameter, in preference to using *class*
  - ♦ It must be used to qualify explicitly any use of types nested within a template parameter type

Declare a type template parameter, *StringType*

```
template<typename StringType>  
    typename StringType::char_type  
    first_non_space(const StringType & string);
```

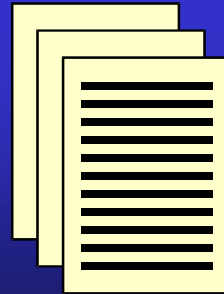
Use a type, *char\_type*, nested in the template type parameter, *StringType*

When templates were first introduced into C++ the principle of keyword economy led to the `class` keyword being reused to indicate a template type parameter, even if the type substituted was a built-in rather than a class type. Later the `typename` keyword was added to language and it was co-opted to serve the same role in a template parameter list. The name `typename` is a more accurate description of the role it plays than `class`, and should be the preferred choice, with `class` considered as legacy. Although it is tempting to use `class` to distinguish parameters that must be user-defined types from `typename` that can indicate any type, such a style has no language support and leads to code with an inconsistent appearance.

The original motivation for including `typename` in the language was to disambiguate names that were resolved in the scope of a template parameter via the scope resolution operator. Until the template is actually instantiated, the compiler cannot in principle tell whether such a nested name is a type or a `static` member. The `typename` keyword is used to clarify when the name is a type name, with all other scoped names assumed to be `static`.

## Abstracting Common Data Structure

- Class templates can avoid duplication where representation differs in a consistent manner
  - ♦ Inheritance can factor out common code that is to be extended but is otherwise invariant
  - ♦ Extends function template model
- Often used for containers and smart pointers
  - ♦ But other uses include traits, policies and metaprogramming

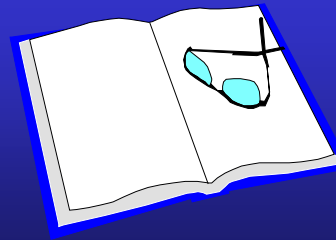


To avoid code bloat from template instantiation of similar code, e.g., `container<foo*>` is likely to have the same binary representation as `container<bar*>`, code that is free of parameterisation can be hoisted into a non-public base class. The derived class template simply acts as an inline wrapper that adapts the types.

Notice also that class templates can have non-type parameters, such as integers (but not floats) and references or pointers to objects with external linkage. For example, `std::bitset` is parameterised on the number of bits it holds.

## Managing Long Type Names

- The use of templates can lead to long and unwieldy type names
  - ♦ An impressive aid to reducing code readability
- *typedefs* can provide shorthand names
  - ♦ Can be block-scoped rather than global
- Template functions deduce the types and remove the need for naming
  - ♦ Used for constructor helpers



Although the syntax for using iterators is simple enough — a subset of pointer notation — the syntax for declaring them almost seems to discourage their use. Consider a list of strings, words, and a function, action, that is applied to each string in the list:

```
for(std::list<std::string>::iterator at = words.begin();
    at != words.end();
    ++at)
    action(*at);
```

There is no way that even the most ardent supporter of generic programming will, with hand on heart, describe the full type declaration as either convenient or elegant. A `using` directive or `using` declaration is not the solution: at best it reduces type names by the amount of `std::`; at worst it negates the benefit of having namespaces in the first place. But a simple tool exists in the language for simplifying this syntax: `typedef`. Although not fully redundant, `typedef` is typically used less in C++ than in C. It is often regarded as the poor cousin of the `class`: it only creates type aliases, rather than proper types with their own distinct behaviour. And this is precisely why it is useful as a local abbreviation facility:

```
typedef std::list<std::string>::iterator input;
for(input at = words.begin(); at != words.end(); ++at)
    action(*at);
```

## *std::pair and std::make\_pair*

*Pairing is commonly used by the library:*

```
template<typename FirstType, typename SecondType>
struct pair
{
    typedef FirstType  first_type;
    typedef SecondType second_type;

    first_type  first;
    second_type second;

    pair();
    pair(const first_type &, const second_type &);
    template<typename OtherFirst, typename OtherSecond>
    pair(const pair<OtherFirst, OtherSecond> &);
};
```

*Constructor helper simplifies use:*

```
template<typename FirstType, typename SecondType>
pair<FirstType, SecondType>
make_pair(const FirstType &, const SecondType &);
```

Extensive use is made of `std::pair` in the standard library:

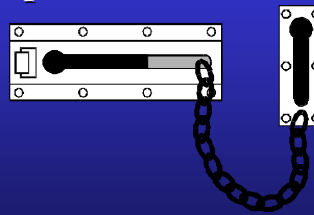
- Returning pairs of iterators that represent an iterator range, e.g., `std::mismatch`.
- Returning a pair containing an iterator and a `bool` from an insertion to a unique key associative container. The `bool` indicates whether a value has been inserted, and the iterator indicates the position.
- The pairing of the key type and the mapped type held in `std::map` and `std::multimap`. This is used for insertion and as the result of iteration.

The `std::make_pair` function template simplifies the construction of paired values:

```
std::pair<std::string, int>(word, occurrence)
std::make_pair(word, occurrence)
```

# Template Compilation and Linkage

- There are two basic models of template compilation...
  - ♦ The *inclusion* model requires template definitions to be visible in header files
  - ♦ The *export* model supports separate compilation
- The *inclusion* model is more portable
  - ♦ The *export* model in its standardised form was never widely supported



The *inclusion* model works like the inline compilation model, and in fact offers better opportunity for compiler-led inlining. Sometimes programmers qualify function templates in headers as `inline` just to be sure the template compilation will do the right thing, but be careful that you do not get what you ask for — inline expansion of long functions causes code bloat, and some compilers blindly follow the `inline` suggestion.

As with inlines, duplicate definitions are eliminated at the link stage. However, the object file size tends to be larger and the compilation time longer.

By contrast, the *export* model was intended to allow separate compilation and defers its build overhead until link time. However, support for the *export* model was thin on the ground and has since been deprecated.

## Dependent Names

- A dependent name is one that depends in some way on a template parameter
  - ♦ E.g., a type nested within template parameter
  - ♦ E.g., a function found by argument dependent lookup based on a template parameter
- The semantics of a dependent name vary between template instantiations
  - ♦ Names looked up both at point of definition and at point of template instantiation



Dependent names are a subtle twist to both the instantiation process and any errors generated. Quoting the C++98 standard:

Inside a template, some constructs have semantics which may differ from one instantiation to another. Such a construct *depends* on the template parameters. In particular, types and expressions may depend on the type and or value of template parameters (as determined by the template arguments) and this determines the context for name lookup for certain names. Expressions may be *typedependent* (on the type of a template parameter) or *valuedependent* (on the value of a nontype template parameter). In an expression of the form:

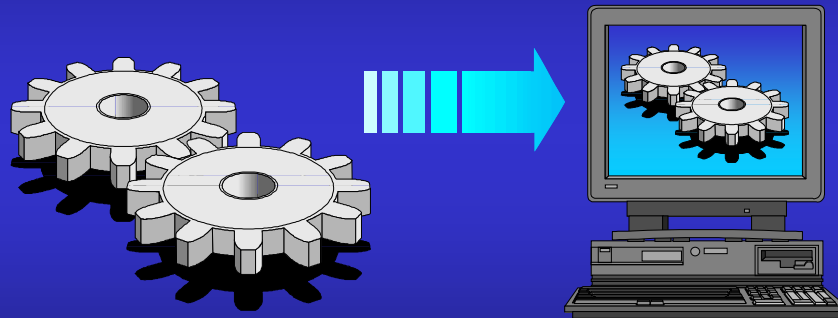
$$postfixexpression \ ( \ expressionlist_{opt} \ )$$

where the *postfixexpression* is an *identifier*, the *identifier* denotes a *dependent name* if and only if any of the expressions in the *expressionlist* is a typedependent expression. If an operand of an operator is a typedependent expression, the operator also denotes a dependent name. Such names are unbound and are looked up at the point of the template instantiation in both the context of the template definition and the context of the point of instantiation.

## Summary

- Templates implement parametric polymorphism in C++
  - ♦ Support conceptual and physical decoupling, although can also introduce physical coupling
- Function templates...
  - ♦ Abstract and factor out common control flow and naming, extending the idea of overloading
- Class templates...
  - ♦ Abstract and factor out common data structure and associated control flow

# Containers



*C++ Advanced*



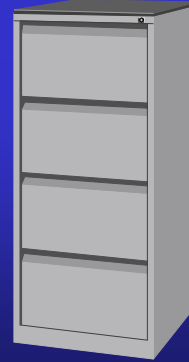
# Containers

- Objectives
  - ◆ Cover the standard container requirements and class templates
- Contents
  - ◆ Containers and requirements
  - ◆ Sequence containers
  - ◆ Associative containers
  - ◆ Containers in future



## Containers and Values

- Containers hold many value objects internally — memory management is encapsulated
  - ♦ *Sequences* organise their elements linearly
  - ♦ *Associative containers* organise elements by key-based lookup
- Standard strings support both container requirements and a more historical interface



Containers are quite lightweight in terms of their specification as they rely mostly on iterators to provide their functionality via external algorithms. A number of parameterising types are specified, such as the reference and pointer types used for the containee type, and then a number of operations that the container must support, e.g., default construction, copy construction, equality and size query. These operations are specified in terms of valid expressions along with their expected behaviour and complexity. For instance, the complexity of the equality operation is linear: the time taken to determine equality of two containers is no worse than proportional to the number of contained elements.

Sequences are a specialised form of container. These are required to satisfy the constraints placed on a container in addition to a number of others, such as insertion and erasing of elements. A sequence is a linear structure based on element position. A number of optional operations are also specified for sequences, such as the subscript operator.

The library provides three proper standard sequence classes. The concept of strings was also been revamped to support a number of sequence operations. To be fair, the `std::basic_string` class template never quite made the grade as a fully-fledged container, but it still warrants discussion as one. It is a little disappointing, but history and evolution had a large part to play in this.

The standard also specifies the requirements, in addition to those for containers, for associative containers. Standard implementations are provided for `set`, `multiset`, `map` and `multimap`. The `map` classes are what are sometimes known as dictionaries or associative arrays, and the `multi` classes are bags.

## Container Type Requirements

- Containers are required to provide a number of publicly accessible types
  - ♦ Elements are *CopyConstructible* and *Assignable*

```
class container
{
public:
    typedef ...          value_type;
    typedef ...          size_type;
    typedef ...          difference_type;
    typedef value_type & reference;
    typedef const value_type & const_reference;
    typedef ...          iterator;
    typedef ...          const_iterator;
    ...
};
```

The types exported by standard container are typically taken from the associated allocator type. There is a restriction in the standard that prevents the reference type from being a smart reference, i.e. a proxy. This is unfortunate as there are cases where this makes sense — generally for custom rather than fully generic standard containers. As it stands these typedefs are required to be actual references, which makes them slightly less useful in the sense that they do not abstract anything.

Iterator types may be nested classes or typedefs of existing types that are built-in, such as pointers, or defined elsewhere, e.g. iterator adaptors. Because the common experience of iterators is that they are nested within the scope of container class templates, it is often assumed that they are secondary abstractions — less important and merely following in the wake of containers. This is certainly not the case, and these turn out to be the most important types on offer.

# Container Member Requirements

All containers are required to support value-based copying

All containers are required to provide support for querying iterator ranges

```
class container
{
public:
    ...
    container();
    container(const container &);
    ~container();
    container & operator=(const container &);
    void swap(container &);

    bool empty() const;
    size_type size() const;
    size_type max_size() const;

    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    ...
};
```

Note that the definition shown above is an *as if* representation, i.e., there is no requirement that the special member functions be defined explicitly or that `bool` results are actually declared `bool`, as opposed to a type convertible to `bool`. However, note that `bool` is used in all standard containers.

`max_size` is a constraint that serves only limited use. It indicates the conceptual maximum size of a container. For most containers this is based on the limit of addressable memory, and for fixed-size containers it will always be the same as `size`.

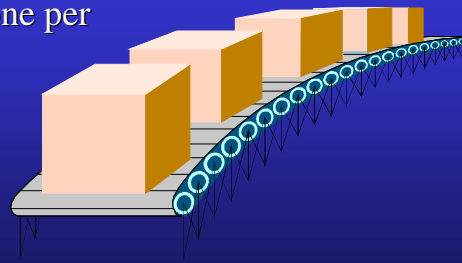
Reversible containers support bidirectional iterators, whereas basic containers are required only to offer forward iterators:

```
class reversible_container
{
public:
    ... // container requirements plus...
    typedef reverse_iterator<const_iterator>
        const_reverse_iterator;
    typedef reverse_iterator<iterator> reverse_iterator;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    ...
};
```

In practice, all of the C++98 standard containers are reversible, but other container types are sometimes non-reversible, e.g., singly linked lists and some hash table implementations.

## Array-based versus Node-based

- Array-based containers...
  - ♦ Store all their elements in one or more blocks of contiguous memory
  - ♦ Support efficient arbitrary access
- Node-based containers...
  - ♦ Store their elements one per chunk of memory
  - ♦ Support efficient arbitrary insertion



For example, an array-based container is something like `std::vector`. This could be implemented as:

```
template<typename ValueType, ...>
class vector
{
    ...
private:
    ...
    ValueType * data;
};
```

Whereas `std::list` is an example of a node-based container:

```
template<typename ValueType, ...>
class list
{
    ...
private:
    ...
    struct link
    {
        link * next, * prior;
        ValueType value;
    };
    link * head, * foot;
};
```

# Sequence Requirements

```
class sequence
{
public:
    ... // container requirements plus...
    sequence(size_type, const value_type &);
    template<typename input_iterator>
        sequence(input_iterator begin, input_iterator end);
    void clear();
    iterator insert(iterator, const value_type &);
    void insert(iterator, size_type, const value_type &);
    template<typename InputIterator>
        void insert(
            iterator, InputIterator, InputIterator);
    iterator erase(iterator);
    iterator erase(iterator from, iterator until);
    ...
};
```

*Sequences are containers that organise their elements linearly.*

Note that insertion is defined only if the iterators that form the source range are not from the container being inserted into.

## Optional Sequence Requirements

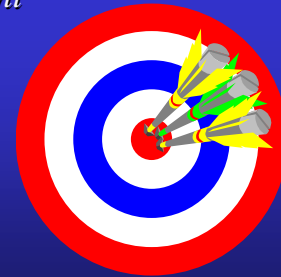
```
class sequence
{
public:
    ... // sequence requirements plus...
    value_type & front();
    const value_type & front() const;
    value_type & back();
    const value_type & back() const;
    void push_front(const value_type &);
    void push_back(const value_type &);
    void pop_front();
    void pop_back();
    value_type & operator[](size_type);
    const value_type & operator[](size_type) const;
    value_type & at(size_type);
    const value_type & at(size_type) const;
    ...
};
```

*These operations are supported only if they take constant time.*

The `at` member function is not that useful and should be overlooked. An interface should represent reasonable goals and present its user with reasonable choices — overachieving interfaces are weaker and more complex, not stronger and simpler. Consider, for instance, the issue of subscripting. `operator[]` is not required to perform bounds checking whereas `at` is. Indexing out of bounds causes undefined behaviour for `operator[]` and an `out_of_range` exception for `at`. On the surface, this looks like a reasonable choice: you get to choose the quality of failure for yourself. The problem is that such an option is useless and cannot be exercised reasonably. When would you consciously choose to write code that needed `at` rather than the more intuitive `operator[]`? If you make the choice, you have already anticipated the bug, and can therefore prevent it.

## *std::vector*

- Resizable array supporting random access
  - ♦ Sequence and reversible container requirements supported
  - ♦ Most optional sequence requirements supported, except for *push\_front* and *pop\_front*
- Internal representation grows exponentially if exceeded by *resize*, *insert* or *push\_back*



A `vector` should generally always be used in preference to a built-in array or an allocated dynamically with `new[]` — particularly the latter. It is more efficient than an array with respect to redundant copying and construction.

Although all implementations implement `vectors` with respect to a single contiguous block of memory, the standard does not require this. This is considered to be a defect and has been fixed (or clarified, depending on your sensibilities) since the original standard.



## *std::vector* Additional Operations

- *capacity* indicates how much space is reserved
  - ♦ *reserve* can be used to avoid reallocations if the capacity can be reasonably anticipated

```
template<typename ValueType, ...>
class vector
{
public:
    ...
    template<typename InputIterator>
        void assign(InputIterator begin, InputIterator end);
    void assign(size_type, const value_type & fill);
    void resize(size_type, value_type fill = value_type());
    size_type capacity() const;
    void reserve(size_type);
    ...
};
```

*capacity* indicates the number of elements that can be accommodated by a vector without reallocating and recopying, whereas *size* indicates the number of constructed objects in the container. This means that *capacity()* - *size()* gives the amount of memory free for expansion, either through *resize* or *push\_back*.

Having spare capacity is what allows *push\_back* to have amortised constant-time complexity, and avoid costly reallocation and copying on every resizing. The optimal automatic growth rate is reckoned to be between 1.5 and 2.5, with many implementations using a factor of 2.

The role of *reserve* is to manually anticipate growth and put aside enough memory for such growth. However, note that there is no corresponding *conserve* member to allow simple shrinkage of capacity. This has to be handled through the gymnastics of making a copy of the container and then swapping the representation between the copy and the original.

## *std::list*

- Doubly-linked list supporting bidirectional iterators with efficient insert and erase
  - ♦ Sequence and reversible container requirements supported
  - ♦ Most optional sequence requirements supported, except for *operator[]* and *at*
  - ♦ Iterators are only invalidated if they refer to elements removed from a list



`std::list` is realised as a fairly typical doubly linked list. Because of its node-based representation, its iterators must be objects rather than built-ins.

## *std::list* Local Operations

```
template<typename ValueType, ...>
class list
{
public:
    ...
    template<typename InputIterator>
        void assign(InputIterator begin, InputIterator end);
    void assign(size_type, const value_type & fill);
    void resize(size_type, value_type fill = value_type());
    void remove(const value_type &);
    template<typename UnaryPredicate>
        void remove_if(UnaryPredicate);
    void unique();
    template<typename BinaryPredicate>
        void unique(BinaryPredicate);
    void sort();
    template<typename BinaryPredicate>
        void sort(BinaryPredicate);
    void reverse();
    ...
};
```

The member functions that have the same name as global algorithm functions should be used in preference to those global functions: they are more efficient.

## *std::list* to *std::list* Operations

- Their node-based nature allows lists to be spliced and merged efficiently
  - ♦ Capabilities that are unavailable on other sequences

```
template<typename ValueType, ...>
class list
{
public:
    ...
    void splice(iterator, list &);
    void splice(iterator, list &, iterator);
    void splice(iterator, list &, iterator, iterator);
    void merge(list &);
    template<typename BinaryPredicate>
        void merge(list &, BinaryPredicate);
    ...
};
```

Splicing and merging two lists is efficient because of the node-based implementation.

## *std::deque*

- Array-based sequence supporting random access and efficient operations at both ends
  - ♦ Sequence and reversible container requirements supported
  - ♦ All optional sequence requirements supported
  - ♦ *resize* and *assign* supported
- Internal representation is page based
  - ♦ I.e. holds array of arrays



The representation of `std::deque` is effectively as follows:

```
template<typename ValueType, ...>
class deque
{
    ...
private:
    ...
    ValueType ** data; // array of arrays
};
```

The double indirection means that iterators must be objects rather than built-ins. For large objects, a deque degenerates to being node-based because the 'pages' are a comparable size to the object (typically 4 kilobytes). The page-based representation also means that iterators are not invalidated for `push_back` or `push_front` operations.

`std::deque` also supports `assign` and `resize` operations, like `std::list` and `std::vector`, but there is no capacity management, as found in `std::vector`.

## *std::basic\_string*

- Resizable string supporting random access
  - ♦ Sequence and reversible container requirements supported
  - ♦ Only a few optional sequence requirements supported, namely *operator[]*, *at* and *push\_back*
- Null characters may be embedded in string

```
template<typename CharType, ...>  
class basic_string;
```

```
typedef basic_string<char>    string;  
typedef basic_string<wchar_t> wstring;
```

The road to Hell is paved with good intentions, and the standard library string type is full of good intentions. The standard `basic_string` class template started life as a simple class and reached its middle age as an over-parameterised class template with a bad name and a bloated interface. The process of standardisation added somewhat more than two cents worth: What about generalisation for internationalisation? What about copy optimisation through reference counting? What about customising its memory allocation? What about safe indexing? What about reverse search operations that mirror each forward search operation? What about support for STL? And support for similar index-based operations? You see, good intentions every one of them. But too much compromise in design leads to a compromised design.

In spite of this criticism, the standard string type is useful. For one thing, it's standard, and for another it satisfies more common string needs than a vanilla `char *`.

## *std::basic\_string* Core Operations

```
template<typename CharType, ...>
class basic_string
{
public:
    ...
    basic_string(const char_type *);
    basic_string & operator=(const char_type *);
    basic_string & operator=(char_type);
    basic_string & operator+=(const basic_string &);
    basic_string & operator+=(const char_type *);
    basic_string & operator+=(char_type);
    void resize(size_type, char_type fill);
    void resize(size_type);
    size_type capacity() const;
    void reserve(size_type);
    const char_type * c_str() const;
    int compare(const basic_string &) const;
    int compare(const char_type *) const;
    ...
};
```

The standard library offers a range of algorithms to make text manipulation code expressive with the minimum of fuss. You can look in the `std::basic_string` class template interface to resolve your string manipulation tasks, but the interface is a large and confused one. Neither complete enough to be considered fully general purpose, nor consistent enough to be considered cohesive. Whilst many basic string processing operations are supported directly within `basic_string`, the core of the STL offers an approach that is more minimal, extensible and consistent. `std::basic_string` satisfies STL container requirements, including the possession of iterators.

`basic_string` has the same growth and reservation model as `vector`. However, there is no requirement that the storage used is contiguous, although that is the typical implementation.

## Interface Issues

- The interface of *basic\_string* is large, inconsistent and incomplete
  - ♦ Duplicates functionality found in algorithms library
  - ♦ Supports two interface models, i.e. some operations are index based and others are iterator based
  - ♦ Closed and not extensible
- The semantics are in places quirky, inconsistent or simply not useful
  - ♦ E.g., *length* versus *size*, *operator[]* versus *at*



The `std::basic_string` interface is cluttered and awkward — both too much and not enough.

It contains a mix of different function styles. There is both legacy from an older non-standardised string design and more recent requirements for STL conformance. This has left some functions using numeric indices and other similar functions using iterators, as well as some wasteful synonyms and near synonyms, e.g., `push_back` and `append` have similar roles but different names, `size` and `length` have the same meaning, the difference between `operator[]` and `at` is more subtle than their exception-handling policies.

Many of `basic_string`'s features that are duplicated in the algorithm library, and serve only to trap the search and replace interface between a rock and a hard place. Neither is it simple and essential enough to be easy to use and consistently extensible, nor is it complete enough to satisfy your more complex string needs.

`basic_string` has three template parameters only one of which, in all honesty, can be said to be useful and usable. The defaulted `char_traits` parameter attempts to solve the wrong problem, and the defaulted `allocator` parameter solves remarkably few problems. The result is that these surplus parameters are a distraction both to library implementers and users, and in all but the rarest uses they are defaulted and ignored... except when they turn up as line noise in compilation error messages.

`basic_string` has an inconsistent approach to quality of failure. Some arguments are validated and lead to thrown exceptions on failure, whereas others just lead to mysterious and undefined behaviour.



## Selecting a Sequence

- There is no such thing as a 'default' sequence
  - ♦ Although the standard proposes that *vector* should be considered as such
- Sequence needs to be chosen according to need
  - ♦ Mostly lookup or mostly insertion?
  - ♦ Insertion and erasing mostly at one end or the other, or in the middle?
  - ♦ Positional lookup or traversal-based usage?
  - ♦ Sorting and searching requirements?



Here is what the C++98 standard says about sequence selection:

A sequence is a kind of container that organizes a finite set of objects, all of the same type, into a strictly linear arrangement. The library provides three basic kinds of sequence containers: `vector`, `list`, and `deque`. It also provides container adaptors that make it easy to construct abstract data types, such as **stacks** or **queues**, out of the basic sequence kinds (or out of other kinds of sequences that the user might define).

`vector`, `list`, and `deque` offer the programmer different complexity trade-offs and should be used accordingly. `vector` is the type of sequence that should be used by default. `list` should be used when there are frequent insertions and deletions from the middle of the sequence. `deque` is the data structure of choice when most insertions and deletions take place at the beginning or at the end of the sequence.

Which, to be frank, is not saying all that much that is useful. For instance, what is meant by 'default'? This wording is more a political than a technical matter.

## Associative Container Requirements

- Associative containers also support a key type and a key comparison type
  - ♦ Strict weak ordering must be used for comparisons
  - ♦ The default is to use less-than ordering

Binary predicate  
function object  
types

```
class associative_container
{
public:
    ... // container requirements plus...
    typedef ... key_type;
    typedef ... key_compare;
    typedef ... value_compare;
    ...
};
```

Associative containers offer content-based organisation. Quoting the C++98 standard:

Associative containers provide an ability for fast retrieval of data based on keys. The library provides four basic kinds of associative containers: `set`, `multiset`, `map` and `multimap`.

Each associative container is parameterized on `Key` and an ordering relation `Compare` that induces a strict weak ordering on elements of `Key`. In addition, `map` and `multimap` associate an arbitrary type `T` with the `Key`. The object of type `Compare` is called the *comparison object* of a container. This comparison object may be a pointer to function or an object of a type with an appropriate function call operator.

The phrase "equivalence of keys" means the equivalence relation imposed by the comparison and *not* the `operator==` on keys. That is, two keys `k1` and `k2` are considered to be equivalent if for the comparison object `comp`, `comp(k1, k2) == false && comp(k2, k1) == false`.

## Associative Core Requirements

- Core requirements accommodate comparisons
  - ♦ Construction and querying

```
class associative_container
{
public:
    ...
    explicit associative_container(
        const key_compare & = key_compare());
    template<typename InputIterator>
        associative_container(
            InputIterator begin, InputIterator end,
            const key_compare & = key_compare());
    void clear();
    key_compare key_comp() const;
    value_compare value_comp() const;
    ...
};
```

The comparison objects returned by the query functions are copies of rather than references to the ones used by the associative container. The comparison criteria used by the container must remain consistent, otherwise the internal data structure can be compromised. Comparison objects that work through an additional level of indirection rather than on just internal values should not allow their ordering criteria to be modified through copies, i.e. where function objects are handle-body objects they should not allow modifiers that affect the body.

## Associative Insertion and Erasure

```
class associative_container
{
public:
    ...
    iterator insert(iterator hint, const_value_type &);
    template<typename InputIterator>
        void insert(InputIterator begin, InputIterator end);
    size_type erase(const key_type &);
    void erase(iterator);
    void erase(iterator from, iterator until);
    ...
};
```

*Plus, for containers with unique keys:*

```
pair<iterator, bool> insert(const value_type &);
```

*Plus, for containers with non-unique keys:*

```
iterator insert(const value_type &);
```

Associative containers that support unique keys can contain at most one element with a particular key. Associative containers supporting non-unique keys may contain multiple elements against the same key.

# Associative Searching

```
class associative_container
{
public:
    ...
    iterator find(const key_type &);
    const_iterator find(const key_type &) const;
    size_type count(const key_type &) const;
    iterator lower_bound(const key_type &);
    const_iterator lower_bound(const key_type &) const;
    iterator upper_bound(const key_type &);
    const_iterator upper_bound(const key_type &) const;
    pair<iterator, iterator> equal_range(const key_type &);
    pair<const_iterator, const_iterator>
        equal_range(const key_type &) const;
    ...
};
```

*Search operations take logarithmic execution time.*

Associative containers are organised according to explicit ordering criteria, which makes them efficient retrieval structures. The member functions provided should be used in preference to non-member algorithms of the same name, i.e. the `find` member executes in logarithmic time for an associative container but in linear time for the global `find` algorithm.

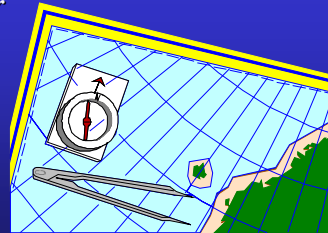
There is no `exists` or `includes` member on associative containers to check for inclusion of a particular key value. However, the functionality already exists in two different forms, `find` and `count`:

- If the key does not exist in the container an iterator to the container's end will be returned, otherwise a valid iterator in the range will be returned. In other words, you check existence by checking that the result of `find` does not equal `end`.
- If the key does not exist in the container the count of the number of times it will occur will equal zero, otherwise it will be greater than zero — at most one in a uniquely keyed container, possibly more in a non-uniquely keyed container. This means that, because of the intimate conversion relationship between `bool` and the rest of the integer types, `count` can be used directly as if it were an existential query.

Note that for uniquely keyed containers either approach has the same performance, whereas for non-uniquely keyed containers the `find` method will typically be more efficient as it terminates after finding the first occurrence.

## *std::map*

- Associative array supporting bidirectional access and unique keys
  - ♦ Associative container and reversible container requirements are supported
  - ♦ Sometimes also known as a 'dictionary'
  - ♦ *value\_type* is key-mapped *pair*
- Key is ordered according to third parameter
  - ♦ Logically represented internally as a binary tree



`std::map` is like a dictionary with unique entries or a relational table.

## *std::map* Insertion

- Two methods for insertion...
  - ♦ Create the associated *value\_type* from key and mapped entry, e.g., use *make\_pair*
  - ♦ Use the just-in-time subscript operation

```
template<typename KeyType, typename MappedType, ...>
class map
{
public:
    ...
    typedef pair<const key_type, mapped_type> value_type;
    ...
    pair<iterator, bool> insert(const value_type &);
    mapped_type &operator[](const key_type &);
    ...
};
```

`std::map` supports two approaches for inserting new elements. One might be considered orthodox and tedious (using `insert`), the other unconventional and convenient (using `operator[]`).

## *std::map* Subscripting

```
typedef std::map<std::string, std::size_t> counter_map;
int main()
{
    counter_map words;
    for(std::string word; std::cin >> word;)
    {
        counter_map::iterator entry =
            words.insert(std::make_pair(word, 0)).first;
        ++entry->second;
    }
    ...
}
```

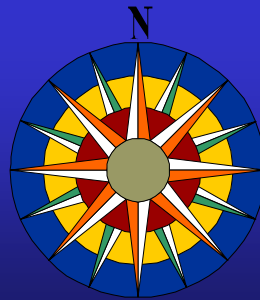
```
int main()
{
    counter_map words;
    for(std::string word; std::cin >> word;)
        ++words[word];
    ...
}
```

The subscripting operator automatically inserts a new element, initialised to the mapped type's default value, if the element being keyed does not already exist. Otherwise it returns the reference to the existing element. The long hand approach, using `insert`, is what it does under the hood.



## *std::multimap*

- Associative array supporting repeating keys
  - ♦ Similar to *map* but with non-unique keys and no magic subscripting ability
  - ♦ Associative container and reversible container requirements are supported
  - ♦ Like a dictionary with multiple entries per key
- Equivalent keys are held in order of insertion



Given that words can have multiple meanings, it could be said that a `multimap` is more like a real-world dictionary than a `map`. It makes no sense for `std::multimap` to support a simple subscript operator, because there is no guarantee of a single value. A case could be made for it to support a more complex subscript operator, one that indexed a proxy element that could be queried for `begin` and `end` for a range of a particular key. However, this functionality is already available through the `equal_range` member function, so the case is not a strong one.

## *std::set*

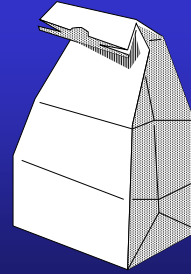
- Associative container supporting unique keys, where the looked-up value is also the key
  - ♦ Associative container and reversible container requirements are supported
  - ♦ *value\_type* is *key\_type*
  - ♦ Bidirectional iterators supported
- Comparisons do not have to involve the whole of the key type
  - ♦ How much of an object is used for comparison depends on the application



The `std::set` holds values uniquely and in ascending order. Potentially a set avoids duplication of key information that may already be in a mapped object.

## *std::multiset*

- Associative container supporting repeating keys, where looked-up value is also the key
  - ♦ Similar to *set* but with non-unique keys
  - ♦ Associative container and reversible container requirements are supported
  - ♦ Sometimes also known as a 'bag'
- As with *multimap*, equivalent keys are held in order of insertion



The `std::multiset` class template rounds out the collection of associative containers, but is probably the least used of all of them.

## Containers in C++11 and Beyond

- Drawn from TR1 and based on Boost, C++11 offers the developer a richer set of containers
  - ♦ Unordered associative containers (i.e. hashed containers): *unordered\_set*, *unordered\_multiset*, *unordered\_map* and *unordered\_multimap*
  - ♦ A fixed-size *array* template that offers an STL-like interface but is not heap based
  - ♦ Ad hoc structures can be supported using the *tuple* template, which can be seen as a generalisation of *std::pair*

The naming convention of `unordered_` avoids clashes with previous implementations that used `hash_`, although it is somewhat clumsy and, retrospectively, suggests that the original associative containers would have been better prefixed with `ordered_`. Hashing is customised through a policy parameter.

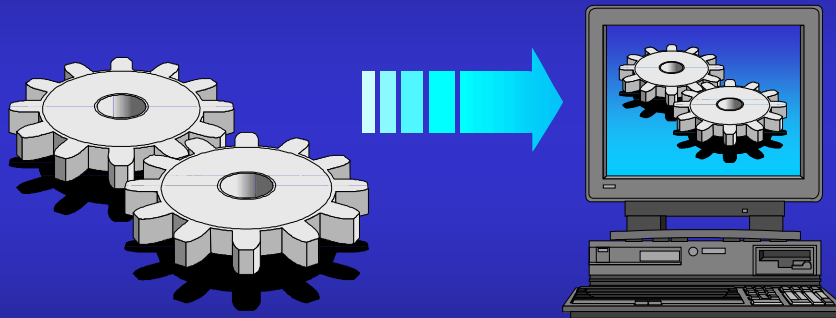
The `array` template is intended for cases where size is statically known and required. By contrast, `std::vector` is intended for homogeneous, resizable containers and does not support aggregate initialisation. And although C-style arrays support aggregate initialisation, they are quite primitive and certainly not STL-like. The `array` template supports aggregate initialisation.

The `tuple` template supports heterogeneous, statically typed, fixed-size containers. It is effectively an ad hoc `struct` whose elements are compile-time indexable. It can be considered a generalisation of `std::pair`, from which it can be assigned and initialised. It also offers comparison operators: ordering is lexicographical, as it is with `pair`.

## Summary

- STL containers come in two parts...
  - ♦ A set of requirements that allow the library to be extensible and open
  - ♦ Concrete container types specified and included as part of the library
- Sequences and associative containers form the core of the container library
  - ♦ Strings are sequences

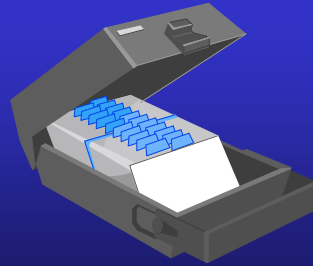
# Iterators



*C++ Advanced*

# Iterators

- Objectives
  - ♦ Examine C++'s iterator model and facilities
- Contents
  - ♦ Iterators and iterator ranges
  - ♦ Iterator categories
  - ♦ *const\_iterator* on containers
  - ♦ Insertion adaptors
  - ♦ I/O stream iterators
  - ♦ Iterator validity



## Iterators and Iterator Ranges

- Iterators are objects for indirect access and traversal of elements in sequence
  - ♦ Traversal of elements without exposing its underlying representation
- Standard C++ iterators are pointer-like
  - ♦ Either raw pointers or smart pointers
  - ♦ Two iterators to the same sequence define a half-open range, i.e., *[begin, end)*



In the STL, iterators are expressed as a generalisation of pointers. Like a pointer, an iterator represents a level of indirection to its target. It supports dereferencing through `*` and `->` operators, and traversal across the elements of its target using pointer arithmetic.

Containers are defined to return iterators to their first element and a notional one-past-the-end element. In the case of containers whose underlying representation is an array, the iterators may simply be typedefs of pointers. This is often the case for implementations of `std::vector` and `std::basic_string`. For other container types, operator overloading provides the appearance of a pointer, e.g. for `std::list`.



## Iterating over a Container

```
typedef std::multiset<std::string> strings;
int main()
{
    strings source;
    for(std::string input; std::cin >> input;)
        source.insert(input);
    strings::iterator at = source.begin();
    while(at != source.end())
        std::cout << *at++ << std::endl;
    return 0;
}
```

*Data read in to `std::multiset` of `std::string` objects, traversed using conventional loop and iterator access.*

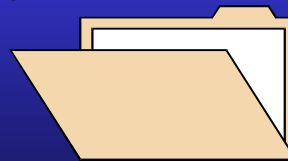
Imagine you want to alphabetically order input to a program. Keeping things simple, let's assume the English alphabet and an ASCII-based character set. The standard `multiset` container uses default *less-than* ordering, and default *less-than* ordering for a string is just what we want. All the input is ordered as it comes in and, because it is a `multiset` rather than a `set`, duplicate strings will be retained.

The subscript operator cannot be used for output because associative containers, such as `multiset`, do not support random access. Iterators offer the means of decoupling traversal of a container from its underlying representation.

Although the condition and body of the loop are quite straightforward, there is potentially a lot of syntactic baggage in the iterator declaration. This load can be lightened a little with the aid of a `typedef`, an oft-neglected feature in a class-based world.

## Iterator Categories

- Iterators are classified by the operations they offer and the traversal they support
  - ♦ *Input iterators* are for single-pass input
  - ♦ *Output iterators* are for single-pass output
  - ♦ *Forward iterators* are for general single-pass
  - ♦ *Bidirectional iterators* allow iteration to and fro
  - ♦ *Random access iterators* support constant-time indexing



The standard specifies five categories of iterator, i.e. five sets of requirements, depending on the kinds of operation supported. With the exception of input and output iterators, each category builds on the category before it. These categories are also represented in the library as tag types which can be used with iterator trait classes to determine the properties and performance of an iterator.

All containers support queries to access start and one past the end iterators in the form of `begin` and `end` member functions. The iterator types for a container are nested within the container class definition: `iterator` for mutable access and `const_iterator` for read-only or browsing access.

# Input Iterators

- Strictly for single-pass reading algorithms
  - ♦ Only small subset of pointer syntax and semantics

```
class input_iterator
{
public:
    input_iterator(const input_iterator &);
    ~input_iterator();
    input_iterator & operator=(const input_iterator &);
    input_iterator & operator++();
    input_iterator operator++(int);
    const value_type & operator*();
    const value_type * operator->();
    bool operator==(const input_iterator &) const;
    bool operator!=(const input_iterator &) const;
    ...
};
```

Input iterators are for single-pass input only. They support copying, pre- and post-increment operators, dereferencing only for reading and basic equality testing. In effect, they are stream-like. And, because you can never twice set foot in the same stream, you cannot expect to revisit values.

Note that the result of equality is required to be convertible to `bool`, and not necessarily `bool` as shown, and likewise the result of dereferencing must be convertible to `value_type`.

## Output Iterators

- Strictly for single-pass writing algorithms
  - ♦ Supports an even smaller subset of pointer syntax and semantics
  - ♦ Common use based on *\*iterator++ = value* syntax

```
class output_iterator
{
public:
    output_iterator(const output_iterator &);
    ~output_iterator();
    output_iterator & operator=(const output_iterator &);
    output_iterator & operator++();
    output_iterator operator++(int);
    value_type & operator*(); // only for assignment
    ...
};
```

Output iterators are for single-pass output only. Their operations are similar to input iterators, except that they support dereferencing only for writing. There is no equality test as you do not test output iterators against a past-the-end output iterator.

Note that as the only use of dereferencing syntax is where the dereferenced result is used on the left-hand side of an assignment, i.e., `*iterator = value`. The requirement for dereferencing is that the result is assignable from a `value_type`. In other words a proxy may be used rather than a `value_type &`, alternatively the iterator type itself may support the assignment directly as part of its own interface. In the latter case, although `iterator = value` could be written, it is not recommended form.

## Forward Iterators

- Extends the input and output iterator categories
  - ♦ Consistent single-pass read and write semantics

```
class forward_iterator
{
public:
    forward_iterator();
    forward_iterator(const forward_iterator &);
    ~forward_iterator();
    forward_iterator & operator=(const forward_iterator &);
    forward_iterator & operator++();
    forward_iterator operator++(int);
    value_type & operator*() const;
    value_type * operator->() const;
    bool operator==(const forward_iterator &) const;
    bool operator!=(const forward_iterator &) const;
    ...
};
```

Forward iterators may be used in multi-pass algorithms, supporting dereferencing for both reading and writing purposes. As you can repass an iterator range, i.e., hold onto an iterator value and reuse it, there is the implication that the underlying range is persistent with respect to iteration. Forward iterators are the simplest iterators that are meaningful for containers. Although the standard does not currently include a container type that offers at best forward iterators, the original STL and many available STLs offer a singly linked list, `slist`. It is likely the next standard will also offer this type.

`const`-ness is not meaningful for input or output iterators, but the dereferenced type of a forward iterator can also be considered either `const` or `non-const`.

## Bidirectional Iterators

- Extends the forward iterator requirements with the ability to back up
  - ♦ Reverse iteration is possible
  - ♦ Values in the iterator range have some permanence

```
class bidirectional_iterator
{
public:
    ... // forward iterator requirements plus...
    bidirectional_iterator & operator--();
    bidirectional_iterator operator--(int);
    ...
};
```

Bidirectional iterators additionally support pre- and post-decrement operators. The standard associative containers — `std::map`, `std::multimap`, `std::set` and `std::multiset` — and `std::list` all offer bidirectional iterators.

## Random Access Iterators

- Nearly full pointer syntax and semantics
  - ♦ In some cases pointers are actually used

```
class random_access_iterator
{
public:
    ... // bidirectional iterator requirements plus...
    random_access_iterator & operator+=(distance_type);
    random_access_iterator operator+(distance_type) const;
    random_access_iterator & operator-=(distance_type);
    random_access_iterator operator-(distance_type) const;
    distance_type operator-(random_access_iterator) const;
    value_type & operator[](distance_type) const;
    bool operator<(const random_access_iterator &) const;
    bool operator<=(const random_access_iterator &) const;
    bool operator>(const random_access_iterator &) const;
    bool operator>=(const random_access_iterator &) const;
    ...
};
```

Random-access iterators support nearly full-blown pointer syntax, including pointer arithmetic and related operations, but not the Boolean and null behaviour.

`std::vector`, `std::deque`, `std::basic_string` and built-in arrays all support random-access iterators. In the case of arrays the iterators are pointers, and in the case of `std::vector` and `std::basic_string` the iterators are often typedefed as pointers.

Note that `operator+` is symmetric, i.e., `iterator + distance` and `distance + iterator`. For brevity, the second form is not shown above.

## *iterator versus const\_iterator*

- The difference between an *iterator* and a *const\_iterator* is one of element access
  - ♦ Not of the operations supported
  - ♦ Both will be of the same category

```
class container::iterator
{
public:
    value_type & operator*() const;
    value_type * operator->() const;
    ...
};
```

```
class container::const_iterator
{
public:
    const value_type & operator*() const;
    const value_type * operator->() const;
    ...
};
```

A non-const function may 'specialise' a function by overloading a const function with a non-const variant. This is most commonly of use where a result should follow qualification, such as overloading the subscript operator for a sequence, or providing iterator accessors for a container. The effect of this idiom is to ensure that a query that returns a part of an object does not grant more privilege to the part than is granted to the whole. For instance, a string class is expected to support subscripting, but it is also expected to be const correct. Thus subscripting a const object should not allow updating:

```
class string
{
public:
    char operator[](size_t) const;
    char & operator[](size_t);
    ...
};
```

The variation in return types is not restricted to values, references and const-ness, e.g.,

```
class container
{
public:
    class iterator;
    class const_iterator;
    iterator begin();
    const_iterator begin() const;
    ...
};
```



## Set Iterators

- Key values for map containers are immutable
  - ♦ They are accessed as *const* and only the mapped value can be modified through an iterator
- But for sets the key *is* the value...
  - ♦ What happens if part of the value needs updating?
  - ♦ Does it make sense to have a non-*const* iterator?
- For portability assume that *iterator* is a *const\_iterator* and keys are totally immutable



Although sets can avoid duplication of information between a mapping key and the object that is mapped, this is not always convenient — an additional level of indirection may upset the simple use of any ordering on the object — and can inconveniently restrict the operations on the object.

## Iteration Adaptors

- Many of the iterator facilities defined in the `<iterator>` header are related to adaptation
  - ♦ Others are related to policy-based programming with iterators
- Adaptors modify appearance and semantics
  - ♦ Iterators made to traverse in reverse
  - ♦ Iterators that work over I/O streams
  - ♦ Iterators over raw memory in object increments



The iterator part of the library also provides adaptors for iterators and iterators that wrap raw memory and I/O streams. The iterator adaptors can change the basic semantics, e.g., traverse backwards, or fundamental syntax, e.g., make a stream appear to be like a container.

## Insertion Iterators

- Container iterators do not rearrange their targeted containers
  - ◆ Common source of errors, e.g., inserting beyond the end of a container or overwriting the beginning
- Insertion iterators adapt container iterators to behave as if they do
  - ◆ Adapted iterators are strictly output iterators

```
std::vector<std::string> words, more;  
...  
std::copy(more.begin(), more.end(), words.end());
```



A common mistake when working with algorithms and containers is to forget that the algorithm is unaware of the container and the iterator has not control over the container. In practice what this means is that elements in the container do not magically appear or move out of the way on what might be intended to be an insertion operation, e.g., attempting to extend a container as the code above shows will have the effect of dereferencing and incrementing a past-the-end iterator. Only madness and debugging will be found in this space.

## Front and Back Insertion Iterators

- *std::front\_insert\_iterator* uses container's *push\_front* member
  - ♦ *std::front\_inserter* constructor helper simplifies use
- *std::back\_insert\_iterator* uses container's *push\_back* member
  - ♦ *std::back\_inserter* constructor helper simplifies use

```
std::vector<std::string> words, more;  
...  
std::copy(  
    more.begin(), more.end(),  
    std::back_inserter(words));
```



The `std::front_insert_iterator` and `std::back_insert_iterator` class templates adapt a container so that output operations magically cause the appearance of new elements at either end. The constructor helpers are normally used, so the `std::front_insert_iterator` and `std::back_insert_iterator` names do not generally appear in relation to your code unless you have compilation errors!

Note that in the particular example above, an alternative and often more appropriate solution for `std::vector` would be as follows:

```
words.insert(words.end(), more.begin(), more.end());
```

## Iterator-Specified Insertion

- *std::insert\_iterator* uses container's single value *insert* member
  - ♦ *std::inserter* constructor helper simplifies use
  - ♦ Both the container and intended insertion position must be specified

```
std::vector<std::string> words, more;  
...  
std::copy(  
    more.begin(), more.end(),           // equivalent to  
    std::inserter(words, words.end())); // back insert
```

An adaptor also exists for inserting from a specified position in a container. Such insertion is intended for sequences and not associative containers.

## Input and Output Stream Iterators

- I/O streams are not containers but can be treated as a single-pass source of values
  - ♦ Either for reading or for assignment
- Stream iterators use `>>` or `<<` on access

```
std::map<std::string, std::size_t> words;
typedef std::istream_iterator<std::string> in_type;
for(in_type in(std::cin); in != in_type();)
    ++words[*in++];
```

```
std::list<std::string> words;
...
std::ostream_iterator<std::string> out(std::cout, "\n");
std::copy(words.begin(), words.end(), out);
```

An `istream_iterator` takes an `istream` object, such as `cin`, and treats it as a sequence of values of its parameter type, in this case `std::string`. The values read in are space separated, and the end of the stream is indicated by a default constructed iterator. Note the use of the `typedef` to simplify local use of an otherwise longwinded name.

Similarly, an `ostream_iterator` takes an `ostream` object, such as `cout`, and treats as a sequence of values that can be assigned to. In effect, it behaves like a write-only pointer.

## *std::streambuf* Iterators

- I/O streams use *streambuf* objects internally
  - ♦ *streambufs* perform character-level buffering and unformatted I/O
- *streambuf* iterators use *sgetc* or *sputc* on access

```
typedef std::istreambuf_iterator<char> in_type;
in_type::difference_type spaces =
    std::count(in_type(std::cin), in_type(), ' ');
```

```
std::deque<char> text;
...
std::ostreambuf_iterator<char> out(std::cout);
std::copy(text.begin(), text.end(), out);
```

*streambuf* iterators work at the level of the character, bypassing the usual I/O formatting of *istream* and *ostream*. This means that they offer a convenient mechanism through to perform I/O for strings represented as sequences other than `char*` or `std::basic_string`.

*streambuf* iterators were added to the library during the process of standardisation and were not part of the original HP STL, which means that they will be missing from some STL implementations. The behaviour of a `std::ostream_iterator` is effectively equivalent, at the character level, to a `std::ostreambuf_iterator`, and so it can be used as a substitute. An `std::istream_iterator` over a typical input stream is normally not equivalent because the stream will eat white space for each formatted input. This feature can be disabled through a flag setting. The following examples are functionally equivalent (although not performance equivalent) to the examples above:

```
typedef std::istream_iterator<char> in_type;
std::cin.unsetf(std::ios::skipws);
in_type::difference_type spaces =
    std::count(in_type(std::cin), in_type(), ' ');
std::deque<char> text;
...
std::ostream_iterator<char> out(std::cout);
std::copy(text.begin(), text.end(), out);
```

## Iterator Validity

- An invalid iterator cannot be dereferenced or used for any purpose, even moving
  - ♦ In node-based containers they only become invalid when their target is erased
  - ♦ In array-based containers they are less stable
- Past-the-end iterators are not dereferenceable or incrementable
  - ♦ Be aware of validity and off-by-one errors when looping and erasing



Attempting to increment an iterator just after the element it refers to has been erased is a common error. It results in undefined behaviour. Either the iterator to the next position needs to be stored before the erase or the result of the `erase` function should be used as the position of the next.

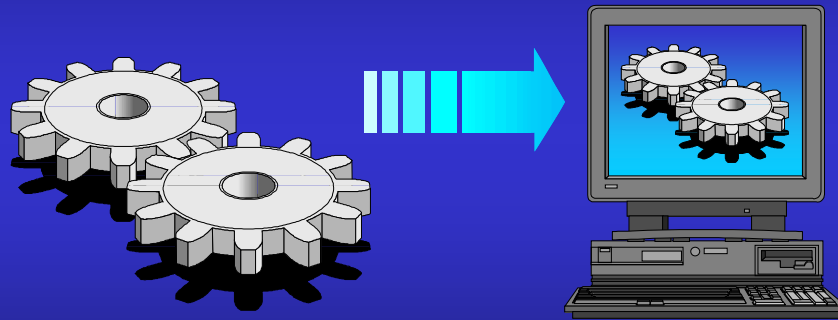
Another common error is of a similar kind and relates to reverse iterators. The `base` of a reverse iterator refers to one off the position dereferenced through the other operations.



## Summary

- Iterators form the bridge between encapsulated algorithms and value-providing data structures
  - ♦ Data structures may be concrete, such as containers, or virtual, such as streams
- Iterators are designed and used according to categories based on syntax and semantics
  - ♦ Substitutable categories offer increasing subsets of full pointer syntax and semantics

# Encapsulated Algorithms



*C++ Advanced*

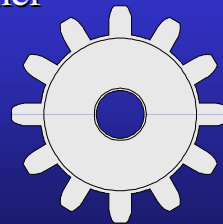
# Encapsulated Algorithms

- Objectives
  - ◆ Present some of the standard function templates
- Contents
  - ◆ Abstracting the *for* loop
  - ◆ Linear searching
  - ◆ Comparing ranges
  - ◆ Numeric algorithms
  - ◆ Replacing and removing elements
  - ◆ Filling and reversing
  - ◆ Sorting and searching



# Algorithms

- The library provides a wealth of function templates that abstract algorithmic control flow
  - ♦ These are based on iterator ranges, a fundamental abstraction of the library
- These templates are commonly referred to as *algorithms*, which is a slight misnomer
  - ♦ An algorithm is a specification of control flow...
  - ♦ The whole point is the encapsulation of the specific algorithm adopted



Although the standard refers to them as algorithms, the main header is called `<algorithm>` and all the STL literature refers to them as algorithms, function templates like `sort` are not algorithms.

An algorithm is a specification of control flow. For instance, Quicksort is an algorithm that specifies a particular way to sort elements in a sequence by recursive partitioning of that sequence. As well as detailing the intent and the performance characteristics, Quicksort is a specification of the steps involved in performing this kind of sort, as opposed to any other sorting algorithm. By contrast, the `std::sort` function template does not specify an algorithm, only the requirements — functional and non-functional — placed on any algorithm that is used to implement it.

## Abstracting the *for* Loop

- The basis of all the algorithmic function templates is to abstract the common *for*
  - ♦ To be precise, a *for* loop over an iterator range
- The simplest example of this is *std::for\_each*
  - ♦ The body of the loop is factored into a separate function or function object

```
template<typename InputIterator, typename UnaryFunction>
UnaryFunction for_each(
    InputIterator, InputIterator, UnaryFunction);
```

Consider the following example:

```
typedef std::list<std::string>::iterator input;
for(input at = words.begin(); at != words.end(); ++at)
    action(*at);
```

Although this example is neater than the equivalent without the `typedef`, we can still do better in terms of readability. Why should we have to know the type `at` at all? Which is the most important feature of this code: the fact that `action` is being applied to each element in `words`; or the loop's housekeeping details? Hopefully the former, but looking at the code suggests that, if real estate is anything to go by, it is the latter. Design is about capturing and reflecting intent:

```
std::for_each(words.begin(), words.end(), action);
```

Just because loops repeat, does not mean that you have to keep rewriting them. The `std::for_each` function template, found in the `<algorithm>` header is perhaps the simplest algorithm in the STL, but it serves to demonstrate the basic idiom of generic programming — the difference between loops for novices and loops for experts.

## Copying Ranges

- Copying from one range to another is one of the most fundamental modification operations
  - ♦ Generalises beyond containers to I/O
  - ♦ Further generalised through iterator adaptors

```
template<typename InputIterator, typename OutputIterator>
OutputIterator copy(
    InputIterator, InputIterator, OutputIterator);
template<typename InBidirectional, OutBidirectional>
OutBidirectional copy_backward(
    InBidirectional, InBidirectional, OutBidirectional);
```

An optimised implementation of the STL should be able to determine that if the source and target iterator ranges are pointers of primitive, bit-wise copyable types, then the copy can be safely implemented using the low-level `memmove` function.

`std::copy_backward` copies downwards from the end of the input range to the output range, descending the output range. It is useful where the input and output ranges overlap and a forward copy would overwrite the targeted locations.

## Linear Searching

- Linear-time search in an iterator range from its beginning until...
  - ♦ Either the first element satisfying the search criterion is reached
  - ♦ Or the end of the range is reached
- The resulting iterator is returned in each case

```
template<typename Iterator, typename ValueType>
bool exists(
    Iterator begin, Iterator end, const ValueType & value)
{
    return std::find(begin, end, value) != end;
}
```

Finding values is one of the commonest uses of looping, and so it should come as no surprise that the STL provides many algorithms in this category. For unsorted ranges a sequential search executes in linear-time.

# Linear Searches on Equality

*Conventional linear search for first equal value in a range:*

```
template<typename InputIterator, typename ValueType>
InputIterator find(
    InputIterator, InputIterator, const ValueType &);
```

*Linear search for first pair of adjacent and equal values in a range:*

```
template<typename ForwardIterator>
ForwardIterator adjacent_find(
    ForwardIterator, ForwardIterator);
```

*Linear search for occurrence in a range of one from a range of values:*

```
template<
    typename ForwardIterator, typename InForwardIterator>
ForwardIterator find_first_of(
    ForwardIterator, ForwardIterator,
    InForwardIterator, InForwardIterator);
```

Equality rather than equivalence is used for the basic search algorithms, finding the first element in a range equal to searched value(s). To search backwards, use reverse iterators.



# Linear Searches on Predicates

*Corresponding versions based on parameterised equality predicates:*

```
template<typename InputIterator, typename UnaryPredicate>
InputIterator find_if(
    InputIterator, InputIterator, UnaryPredicate);
```

```
template<
    typename ForwardIterator, typename BinaryPredicate>
ForwardIterator adjacent_find(
    ForwardIterator, ForwardIterator, BinaryPredicate);
```

```
template<
    typename ForwardIterator, typename InForwardIterator,
    typename BinaryPredicate>
ForwardIterator find_first_of(
    ForwardIterator, ForwardIterator,
    InForwardIterator, InForwardIterator,
    BinaryPredicate);
```

The following example finds the first occurrence of a null pointer in a container of pointers:

```
std::deque<task *> tasks;
...
std::deque<task *> found =
    std::find(
        tasks.begin(), tasks.end(),
        std::logical_not<task *>());
```

# Linear Searches for Least and Most

*Search for the first occurrence of the minimum element in a range:*

```
template<typename ForwardIterator>
ForwardIterator min_element(
    ForwardIterator, ForwardIterator);
template<..., typename BinaryPredicate>
ForwardIterator min_element(..., BinaryPredicate);
```

*Search for the first occurrence of the maximum element in a range:*

```
template<typename ForwardIterator>
ForwardIterator max_element(
    ForwardIterator, ForwardIterator);
template<..., typename BinaryPredicate>
ForwardIterator max_element(..., BinaryPredicate);
```

In conjunction with `swap`, the `min_element` function template can be used to implement a selection sort algorithm quite concisely:

```
template<typename iterator>
void sort(iterator begin, iterator end)
{
    for(; begin != end; ++begin)
        std::swap(*begin, *std::min_element(begin, end));
}
```

Selection sort has the same performance complexity as its infamous cousin, the bubble sort, but is somewhat more compact, presentable and easier to get right.

## Comparing Ranges

- Iterator ranges can be compared against one another in sequence
  - ♦ For similarity, dissimilarity or ordering
- A resulting comparison or position is returned

```
template<
    typename FirstIterator, typename SecondIterator>
bool equivalent(
    FirstIterator begin1st, FirstIterator end1st,
    SecondIterator begin2nd, SecondIterator end2nd)
{
    return !std::lexicographical_compare(
        begin1st, end1st, begin2nd, end2nd) &&
        !std::lexicographical_compare(
            begin2nd, end2nd, begin1st, end1st);
}
```

The `equivalent` algorithm shown above is not part of the standard, but demonstrates how to compare two ranges for ordering equivalence, as opposed to equality, based on their lexicographical ordering. Recall that the definition of ordering equivalence for two values `a` and `b` is `!(a < b) && !(b < a)`.

## Range Comparisons

*Compare two ranges for equality or order, or find first equality mismatch:*

```
template<typename LhsInput, typename RhsInput>
    bool equal(LhsInput, LhsInput, RhsInput);
template<..., typename BinaryPredicate>
    bool equal(..., BinaryPredicate);
```

```
template<typename LhsInput, typename RhsInput>
    bool lexicographical_compare(
        LhsInput, LhsInput, RhsInput, RhsInput);
template<..., typename BinaryPredicate>
    bool lexicographical_compare(..., BinaryPredicate);
```

```
template<typename LhsInput, typename RhsInput>
    pair<LhsInput, RhsInput> mismatch(
        LhsInput, LhsInput, RhsInput);
template<..., typename BinaryPredicate>
    pair<LhsInput, RhsInput> mismatch(..., BinaryPredicate);
```

Lexicographical comparison is effectively dictionary comparison, i.e., each element in sequence until either the elements differ between the two ranges or one of the ranges runs out.

## Associative Container Algorithms

- Associative containers support many member functions similar to non-member algorithms
  - ♦ E.g. *find* and *count*
- For efficiency and brevity, prefer the member to the non-member versions

```
std::set<std::string> words;  
std::string word;
```

```
... find(words.begin(), words.end(), word) != words.end();  
... count(words.begin(), words.end(), word);
```

```
... words.find(word) != words.end();  
... words.count(word);
```



Equivalently named search member functions on associative containers will complete the search in logarithmic execution time rather than linear. So although the linear version will work, it is suboptimal.

## Numeric Algorithms

- There are a number of algorithms dedicated to common numeric operations on ranges
  - ♦ *std::accumulate*, *std::inner\_product*, *std::partial\_sum* and *std::adjacent\_difference*
- Can sometimes also be used for non-numerics

```
template<typename InputIterator, typename ValueType>
ValueType accumulate(
    InputIterator, InputIterator, ValueType);
template<..., typename BinaryFunction>
ValueType accumulate(..., BinaryFunction);
```

These algorithm function templates are found in the `<numeric>` header rather than in the `<algorithm>` header, although historically that is where they originated.

# Accumulating

```
std::vector<double> data;
...
double average =
    std::accumulate(
        data.begin(), data.end(), 0.0) / data.size();
```

*Summing numbers is a clear use, but can be used for string manipulation:*

```
std::string paste(
    const std::string & lhs, const std::string & rhs)
{
    return lhs + " " + rhs;
}

int main(int argc, char * argv[])
{
    std::string args =
        std::accumulate(
            argv + 1, argv + argc, std::string(), paste);
    ...
}
```

A common gotcha using the `accumulate` algorithm is to forget that the type of the accumulated value is based on the initial value given. This means that had the call in the first example above been written as

```
std::accumulate(data.begin(), data.end(), 0)
```

The accumulation would have been performed using an `int` initialised to 0 rather than a `double` initialised to 0.0. With templates argument deduction, many of the common and convenient implicit conversions that you may have become accustomed to are no longer applicable.

# Transformation

- *std::transform* is similar to *std::for\_each*
  - ♦ However, it writes function invocation results to an output iterator rather than discarding them

```
template<
    typename InputIterator, typename OutputIterator,
    typename UnaryFunction>
OutputIterator transform(
    InputIterator, InputIterator, OutputIterator,
    UnaryFunction);

template<
    typename LhsInputIterator, typename RhInputIterator,
    typename OutputIterator, typename BinaryFunction>
OutputIterator transform(
    LhsInputIterator, LhsInputIterator,
    RhInputIterator, OutputIterator, BinaryFunction);
```

The end of the output iterator sequence that `transform` writes to is returned, whereas the (often ignored) result from `for_each` is a copy of the unary function pointer or function object that was used. In the case of `transform`, the function or function object used often acts like a pure function, whereas a function object given to `for_each` may accumulate results that the caller wishes to pick up.



## *std::transform versus std::for\_each*

```
void quoted(const std::string & text)
{
    std::cout << '"' << text << '"' << std::endl;
}
int main(int argc, char * argv[])
{
    std::for_each(argv + 1, argv + argc, quoted);
    ...
}
```

```
std::string quoted(const std::string & text)
{
    return '"' + text + '"';
}
int main(int argc, char * argv[])
{
    typedef std::ostream_iterator<std::string> out;
    std::transform(
        argv + 1, argv + argc, out(std::cout, "\n"), quoted);
    ...
}
```

In this particular example the style of function used with `transform` is a purer functional form than the style used with `for_each`, which relies more explicitly on the function having a side effect, i.e., output.

## Replacing Elements in a Range

- Elements in a range may be replaced in place or copied to another range if...
  - ♦ A value is selected for replacement by an equality comparison
  - ♦ A value is selected for replacement by a predicate

```
std::replace(text.begin(), text.end(), '-', ' ');
```

```
std::replace_if(  
    text.begin(), text.end(), std::isspace, '_');
```

A common data processing task is the simple transformation of one format into another, or the reduction of various optional presentation forms to a common one. For example, ISBNs can be listed with space separation between the number groups or with hyphens. Input can be processed so that all ISBNs that a program works with internally use only spaces. The first code fragment above shows how `replace` can be used to replace all occurrences of a hyphen with a space.

# Replacement

```
template<typename ForwardIterator, typename ValueType>
void replace(
    ForwardIterator, ForwardIterator,
    const ValueType & from, const ValueType & to);
template<..., typename UnaryPredicate, typename ValueType>
void replace_if(
    ForwardIterator, ForwardIterator,
    UnaryPredicate, const ValueType & to);
```

```
template<
    typename InputIterator, typename OutputIterator,
    typename ValueType>
OutputIterator replace_copy(
    InputIterator, InputIterator, OutputIterator,
    const ValueType & from, const ValueType & to);
template<..., typename UnaryPredicate, typename ValueType>
OutputIterator replace_copy_if(
    InputIterator, InputIterator, OutputIterator,
    UnaryPredicate, const ValueType & to);
```

The `_copy` variants of `replace` leave the original sequence untouched, and so could be considered more functional style.

## Removing Elements from Containers

- From associative containers...
  - ♦ Remove elements of a particular value with an *erase* member function
- From *std::list*...
  - ♦ Use the *remove* and *remove\_if* member functions
- From other sequences...
  - ♦ Use the non-member algorithms followed by *erase*

```
text.erase(  
    std::remove(text.begin(), text.end(), ' '),  
    text.end());
```

To handle the removal of characters the `std::remove` algorithm seems like the obvious candidate. It is part of the solution, but it comes with a twist. The following code almost does what we want:

```
std::remove(text.begin(), text.end(), ' ');
```

The twist is that in spite of its name, the `std::remove` algorithm does not actually remove elements from the target container itself. Remember that algorithms operate on iterators not containers. The `remove` algorithm does effectively remove the given value from the range but it does not — and cannot — change the range itself. Its return value is the end of the resulting range once all the values have been shunted up, which means that the given string needs to be resized to eliminate the leftovers:

```
std::string::iterator new_end =  
    std::remove(text.begin(), text.end(), ' ');  
text.erase(new_end, text.end());
```

Or, more concisely as a single statement, following the principle of letting the compiler do the type deduction work. The standard does not specify what is in the junk left at the end of the range after removal, so ignoring or erasing it is about the only reasonable thing you can do.

## Removing Elements from a Range

```
template<typename ForwardIterator, typename ValueType>
ForwardIterator remove(
    ForwardIterator, ForwardIterator, const ValueType &);
template<typename ForwardIterator, typename UnaryPredicate>
ForwardIterator remove_if(
    ForwardIterator, ForwardIterator, UnaryPredicate);
```

```
template<
    typename InputIterator, typename OutputIterator,
    typename ValueType>
OutputIterator remove_copy(
    InputIterator, InputIterator,
    OutputIterator, const ValueType &);
template<
    typename InputIterator, typename OutputIterator,
    typename UnaryPredicate>
OutputIterator remove_copy_if(
    InputIterator, InputIterator,
    OutputIterator, UnaryPredicate);
```

Another common data processing need is to compact output by filtering out certain values. For example, removing spaces from ISBNs so that only a sequence of digits is left. This role can be fulfilled by the `std::remove` algorithm.

## Removing Adjacent Duplicates

```
template<typename ForwardIterator>
ForwardIterator unique(
    ForwardIterator, ForwardIterator);

template<
    typename ForwardIterator, typename BinaryPredicate>
ForwardIterator unique(
    ForwardIterator, ForwardIterator, BinaryPredicate);
```

```
template<
    typename InputIterator, typename OutputIterator>
OutputIterator unique_copy(
    InputIterator, InputIterator, OutputIterator);

template<
    typename ForwardIterator, typename OutputIterator,
    typename BinaryPredicate>
OutputIterator remove_copy_if(
    InputIterator, InputIterator,
    OutputIterator, BinaryPredicate);
```

Note that `std::list` supports its own `unique` and `unique_if` on member functions, and these should be used in preference to the non-member algorithms.

Also note that the same idiom that is used for removing the tail end of a sequence after calling `std::remove` must be applied for `std::unique`.

## Filling Ranges

```
template<typename ForwardIterator, typename ValueType>
void fill(
    ForwardIterator, ForwardIterator, const ValueType &);
template<
    typename OutputIterator, typename CountType,
    typename ValueType>
void fill_n(
    ForwardIterator, CountType, const ValueType &);
```

*The function or function object is called to provide for each element:*

```
template<typename ForwardIterator, typename NullaryFunction>
void generate(
    ForwardIterator, ForwardIterator, NullaryFunction);
template<
    typename OutputIterator, typename CountType,
    typename NullaryFunction>
void generate_n(
    OutputIterator, Count_type, NullaryFunction);
```

It is possible to fill a specified range with a single value repeated or a value generated by a function or function object. This role can sometimes be filled (sic) by `resize` member functions and constructors on sequence containers.

## Reversing Ranges and Containers

- Sequences can be reversed in place or copied in reverse order
  - ♦ The *reverse* member function should be used for reversing *std::list* objects

```
template<typename BidirectionalIterator>
void reverse(
    BidirectionalIterator, BidirectionalIterator);
template<
    typename BidirectionalIterator,
    typename OutputIterator>
OutputIterator reverse_copy(
    BidirectionalIterator, BidirectionalIterator,
    OutputIterator);
```

Note that `std::list` supports its own `reverse` member function, and this should be used in preference to the non-member algorithm.



## Sorting Sequences

- Random access sequences are sorted by external algorithms
  - ♦ Associative containers are already sorted
  - ♦ *std::list* supports its own *sort* members

```
std::vector<std::string> words;  
...  
std::sort(words.begin(), words.end());  
...  
std::sort(  
    words.begin(), words.end(),  
    std::greater<std::string>());
```

The following is an implementation of `is_sorted`, which is useful but not supported as standard within the STL:

```
template<typename const_iterator>  
bool is_sorted(const_iterator begin, const_iterator end)  
{  
    if(begin != end)  
    {  
        const_iterator previous;  
        do  
        {  
            previous = begin++;  
        }  
        while(begin != end && !(*begin < *previous));  
    }  
    return begin == end;  
}
```

## Full Sorting

- A complete sort can either retain or discard the original order of equivalent elements
  - ♦ By default less-than ordering is used

```
template<typename RandomAccessIterator>
void sort(
    RandomAccessIterator, RandomAccessIterator);
template<..., typename BinaryPredicate>
void sort(..., BinaryPredicate);
```

```
template<typename RandomAccessIterator>
void stable_sort(
    RandomAccessIterator, RandomAccessIterator);
template<..., typename BinaryPredicate>
void stable_sort(..., BinaryPredicate);
```

`std::stable_sort` retains relative ordering of equivalent elements, and so will take either more time or more memory than a plain `sort`. It will take at most  $O(N (\log N)^2)$  operations but, if dynamic memory is available, this can be reduced to  $O(N \log N)$ .

# Partial Sorting

```
template<typename RandomAccess>
void partial_sort(
    RandomAccess begin, RandomAccess middle,
    RandomAccess end);
template<..., typename BinaryPredicate>
void partial_sort(..., BinaryPredicate);
```

```
template<typename Input, typename RandomAccess>
void partial_sort_copy(
    Input begin, Input end,
    RandomAccess begin_out, RandomAccess end_out);
template<..., typename BinaryPredicate>
void partial_sort_copy(..., BinaryPredicate);
```

```
template<typename RandomAccess>
void nth_element(
    RandomAccess, RandomAccess nth, RandomAccess);
template<..., typename BinaryPredicate>
void nth_element(..., BinaryPredicate);
```

`std::partial_sort` does not sort a whole container. It finds the middle – begin elements that would be at the beginning of the sorted range `[begin, end)`, and places them in sorted order in `[begin, middle)`. The remaining elements are in an unspecified order.

`std::nth_element` is an even more limited form of partial sorting: the position pointed to by `nth` will be the element in the right position if the whole range had been sorted.

# Partitioning

- Partitioning is a very partial sorting indeed
  - ◆ A range is divided into two according to a given predicate
  - ◆ All elements that satisfy the predicate are placed before all the elements that do not

```
template<typename BidirectionalIterator>
    BidirectionalIterator partition(
        BidirectionalIterator, BidirectionalIterator,
        BinaryPredicate);
template<typename BidirectionalIterator>
    BidirectionalIterator stable_partition(
        BidirectionalIterator, BidirectionalIterator,
        BinaryPredicate);
```

The returned iterator points to the partitioning element.

## Searching Sorted Sequences

- Searching on a sorted range does not require linear searching
  - ♦ Binary searches are logarithmic in executing comparisons rather than sequential and linear
- When searching for a given value, equivalence rather than equality is used

```
template<typename ForwardIterator, typename ValueType>
bool binary_search(
    ForwardIterator, ForwardIterator, const ValueType &);
template<..., typename BinaryPredicate>
bool binary_search(..., BinaryPredicate);
```

As it returns only `true` or `false`, the `binary_search` algorithm is of limited use if you want to work with the found value. However, it can usefully play the more modest role of testing set membership.

## Lower and Upper Bounds

```
template<typename ForwardIterator, typename ValueType>
ForwardIterator lower_bound(
    ForwardIterator, ForwardIterator, const ValueType &);
template<..., typename BinaryPredicate>
ForwardIterator lower_bound(..., BinaryPredicate);
```

```
template<typename ForwardIterator, typename ValueType>
ForwardIterator upper_bound(
    ForwardIterator, ForwardIterator, const ValueType &);
template<..., typename BinaryPredicate>
ForwardIterator upper_bound(..., BinaryPredicate);
```

```
template<typename ForwardIterator, typename ValueType>
pair<ForwardIterator, ForwardIterator> equal_range(
    ForwardIterator, ForwardIterator, const ValueType &);
template<..., typename BinaryPredicate>
... equal_range(..., BinaryPredicate);
```

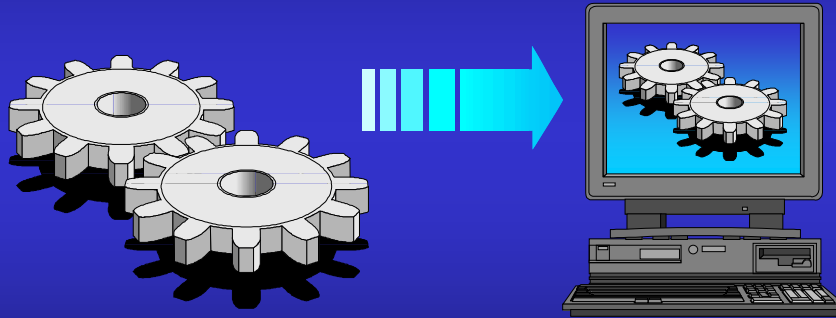
On failure, these algorithms return iterators to where the values would have been if they were included in the sorted range.

Never call `lower_bound` followed by `upper_bound`, use `equal_range` instead.

## Summary

- Algorithm-based function templates make up the engine of STL
  - ◆ They drive or are driven by collections of data, whether concrete or virtual, through iterators
  - ◆ They form a decoupled and extensible family of common operations
- Algorithms represented directly in containers are preferred to the non-member counterparts

# Function Objects

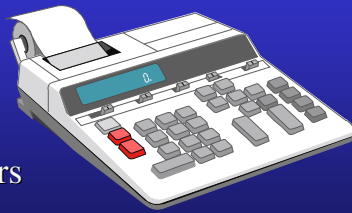


*C++ Advanced*



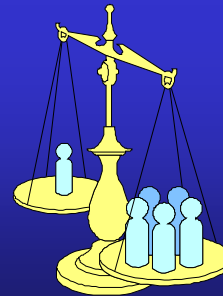
# Function Objects

- Objectives
  - ◆ Define the standard function object model
- Contents
  - ◆ Function object usage and anatomy
  - ◆ Classifying functions and function objects
  - ◆ Adaptable function objects
  - ◆ Standard function objects
  - ◆ Customising associative containers
  - ◆ Binders, negators and adaptors



## Functions versus Function Objects

- A function cannot retain state, overloading and templating must be resolved before passing
  - ♦ A function pointer is used for callbacks, which means that inline functions are not expanded
- A function object can retain state and can be more generic
  - ♦ An inlined implementation will be expanded as part of the templated algorithm



A common piece of advice offered to developers making a transition from procedural to object-oriented code is that a class should not model a function. Such classes are often named as actions, and typically sport a principal or single member function named "do such and such". While this advice does guard against a common pitfall, it is not always poor practice. Those that have taken this rule of thumb to heart as a legalistic rule need to unlearn a little to appreciate how objects can encapsulate tasks and, in particular, mimic functions. The Command pattern demonstrates the power of task-based objects. The Functor or Function Object idiom focuses on functional objects that overload `operator()` to achieve the appearance and transparency of use of conventional functions.

Perhaps surprisingly function objects can often be more efficient than the equivalent functions. This is mainly down to modern compiler technology and the selection of what gets inlined and what does not — it could change in future. What it means in the short term is that it is possible to write C++ code that is measurably more efficient than the corresponding C code.

Another important efficiency consideration is copying and copyability. Function objects must be *CopyConstructible*, and it is best if they are lightweight. Often they can rely on the defaulted copy constructor, i.e., 'copyright'.

# Functions with Algorithms

```
class task
{
public:
    virtual ~task();
    ...
};
```

*Function that is designed to act on individual items or be used in iteration over many:*

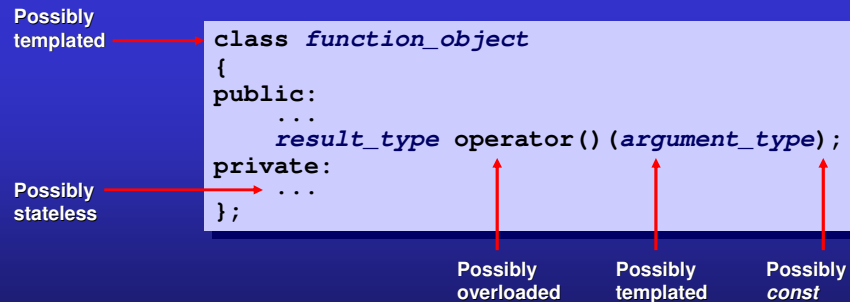
```
void task_deleter(task *ptr)
{
    delete ptr;
}
```

```
std::deque<task *> tasks;
...
std::for_each(tasks.begin(), tasks.end(), task_deleter);
```

The simplest form of behavioural parameterisation for an algorithm function is through a function or, to be precise, a function pointer. This callback approach is most reminiscent of C in its style, but works equally — and often more safely with respect to type — in C++.

# Function Object Anatomy

- The essential feature of any function object is its overloading of the function-call operator
  - ♦ However, there are many variations beyond that



Where function objects have single argument constructors, they should be declared `explicit` as a converting constructor makes little sense.

It is common to implement stateless function object types as `structs`, given that access neither reveals nor hides anything useful about such types. In such cases the `operator()` should be declared `const` as there is nothing to change.

# Function Objects with Algorithms

*Function object that retains state over multiple uses:*

```
struct task_deleter
{
    task_deleter() : deleted(0) {}
    void operator()(task *ptr)
    {
        if(ptr)
        {
            delete ptr;
            ++deleted;
        }
    }
    std::size_t deleted;
};
```

```
std::size_t tasks_deleted =
    std::for_each(
        tasks.begin(), tasks.end(),
        task_deleter()).deleted;
```

The `task_deleter` type is shown here as a `struct` with `public` data for brevity of exposition, rather than as a recommendation of good practice. However, the point about statefulness is clear: the function object accumulates a value over the course of the iteration.

## Nullary, Unary and Binary

- Nullary functions and function objects...
  - ♦ Are called with zero arguments
  - ♦ Are not officially classified in the standard
- Unary functions and function objects...
  - ♦ Are called with one argument
- Binary functions and function objects...
  - ♦ Are called with two arguments
- Higher orders are typically not classified

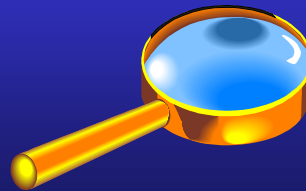


The standard library provides for the use of function objects with generic functions and templated containers, categorising them as unary or binary functions. It also defines specific function object classes — e.g., `less` for ordered comparison — and function object adaptors — e.g., `pointer_to_unary_function` to wrap up naked function pointers. The Boost library extends this with other function object classes, adaptors, and the nullary function category, for function objects taking no arguments.

Although nullaries are not officially recognised by the standard, the STL does offer generators, which take no arguments and return a value.

## Predicates

- Predicates are unary or binary functions or function objects that return a truth value
  - ♦ Either *bool* or something convertible to *bool*
- No changes to the argument(s) are permitted, and side effects in general should be avoided
  - ♦ Both the argument and the function object should be considered *const*



Predicates should be pure queries — pure functions in the mathematical and functional programming sense. In other words, the effect of calling a given predicate object or function with the same arguments should be the same whenever the call is made.

## Adaptable Function Objects

- With the appropriate interface function objects can be combined or adapted to other forms
  - ♦ Composability is a key strength of function objects

```
struct unary_function_object
{
    typedef argument_type argument_type;
    typedef result_type    result_type;
    ...
};
```

```
struct binary_function_object
{
    typedef first_argument_type  first_argument_type;
    typedef second_argument_type second_argument_type;
    typedef result_type          result_type;
    ...
};
```

Adaptable function objects export trait information in their interfaces so that an adaptor, a function or other object can make use of knowledge of the return type and argument types.



# Convenience Base Class Templates

*Base classes intended solely to provide appropriate typedefs:*

```
template<typename ArgumentType, typename ResultType>
struct unary_function
{
    typedef ArgumentType argument_type;
    typedef ResultType    result_type;
};
```

```
template<
    typename FirstArgumentType,
    typename SecondArgumentType,
    typename ResultType>
struct binary_function
{
    typedef FirstArgumentType  first_argument_type;
    typedef SecondArgumentType second_argument_type;
    typedef ResultType        result_type;
};
```

To simplify the implementation of adaptable function objects, the `std::unary_function` and `std::binary_function` base class templates are provided. Such inheritance is for convenience only, and there is no intent that the such function objects should be passed or manipulated via pointers or references to their base class.

## From Operators to Function Objects

- A function object type exists for almost all the common unary and binary C++ operators
  - ♦ Arithmetic, logical and relational operations
- The function object types are all templated, stateless and adaptable
  - ♦ They derive from *unary\_function* and *binary\_function* base templates as appropriate
  - ♦ The comparison and logical types are all predicates



Many of the built-in operators — unary and binary arithmetic, relational and logical side-effect-free operators — have corresponding function object types in the library. These allow basic customisable arithmetic, comparison or logical selection to be performed with standard algorithms.

# Arithmetic Function Object Types

```
template<typename Type>
struct negate : unary_function<Type, Type>
{
    Type operator() (const Type &) const;           // -
};
```

*The binary function object types all have a similar definition:*

```
template<typename ArithmeticType> struct plus;      // +
template<typename ArithmeticType> struct minus;    // -
template<typename ArithmeticType> struct multiplies; // *
template<typename ArithmeticType> struct divides;  // /
template<typename ArithmeticType> struct modulus;  // %
```

```
template<typename Type>
struct arithmetic_function : binary_function<Type, Type, Type>
{
    Type operator() (const Type & lhs, const Type & rhs) const;
};
```

Note that although there is a `negate` function object type, there is no `identity`, although it is included as a non-standard feature in some STL implementations.

# Logical Function Object Types

```
template<typename LogicalType>
struct logical_not : unary_function<LogicalType, bool>
{
    bool operator() (const LogicalType &) const;          // !
};
```

*The binary function object types all have a similar definition:*

```
template<typename LogicalType> struct logical_and;      // &&
template<typename LogicalType> struct logical_or;      // ||
```

```
template<typename Type>
struct logical_function : binary_function<Type, Type, bool>
{
    bool operator() (const Type & lhs, const Type & rhs) const;
};
```

Logical function object types, in particular the binary predicates, allow useful classifications and transformations of Boolean data. They are particularly effective when used in combination with binders and negators.

## Comparison Function Object Types

- The comparison function object types all have a similar definition
  - ♦ Used for equality, equivalence or ordering

```
template<typename ComparableType> struct equal_to;    // ==
template<typename ComparableType> struct not_equal_to; // !=
template<typename ComparableType> struct less;       // <
template<typename ComparableType> struct less_equal; // <=
template<typename ComparableType> struct greater;    // >
template<typename ComparableType> struct greater_equal; // >=
```

```
template<typename Type>
struct comparison_function : binary_function<Type, Type, bool>
{
    bool operator()(const Type & lhs, const Type & rhs) const;
};
```

The comparison function object types are perhaps the most commonly used. For instance, the following is a container that orders its unique keys in descending order:

```
std::set< std::string, std::greater<std::string> > > names;
```

## Customising Associative Containers

- The standard associative containers have a binary predicate function object type parameter
  - ♦ Used for comparison of elements based on strict weak ordering
- Alternative orderings may be parameterised
  - ♦ Default sorting for pointers may not be appropriate

```
std::multimap<
    int, std::string, // marks and student name
    std::greater<int> // highest marks first
> marks;
```

The default ordering for elements is through `std::less`. For pointers this is required to give a total ordering, i.e., emulate a flat memory model if one does not exist. However, although it will support uniqueness with respect to object identity, it does not give a useful order or truly content-based organisation.

## Function Object Customisation

- Provide a dereferencing function object type
  - ♦ Appropriate comparison is based on predicate provided rather than the key type

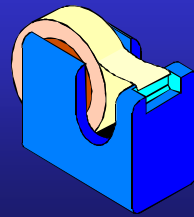
```
struct c_str_less
{
    bool operator()(const char * lhs, const char * rhs) const
    {
        return strcmp(lhs, rhs) < 0;
    }
};
```

```
std::map<const char *, symbol, c_str_less> symbols;
```

By default, the ordering on `std::map<const char *, symbol>` will be arbitrary, based on ordered pointer identity rather than on lexical ordering of content identity. A simple function object type, based on the ordering provided by `strcmp`, can provide the map with the appropriate ordering.

## Binders

- A binder remembers a value and a binary function object
  - ♦ A binder calls its remembered function object with its remembered value and its given argument
  - ♦ Constructor helpers greatly simplify usage
- Except for the position bound, *binder1st* and *binder2nd* are similarly defined
  - ♦ Likewise *bind1st* and *bind2nd* differ only with respect to the binder object type they return



Binders effectively allow function objects to be bound to a given argument value, effectively a form of currying. For instance, in the following example, the search criteria is to find the first positive number in a sequence of data:

```
std::list<int> data;  
...  
std::list<int>::iterator found =  
    std::find(  
        data.begin(), data.end(),  
        std::bind2nd(std::greater<int>(), 0));
```



# Negators

- Negators take a predicate and return the complement of their result
  - ♦ Often used in conjunction with binders
  - ♦ Exist both in unary and binary forms

```
template<typename InputIterator, typename ValueType>
InputIterator find_not(
    InputIterator begin, InputIterator end,
    const ValueType & value)
{
    return std::find_if(
        begin, end,
        std::not1(
            std::bind2nd(
                equal_to<ValueType>(), value)));
}
```

Where a search looks for something that *is* the case, a negator allows the criteria of the search to be inverted, i.e., to find something that is *not* the case, whilst still using the same search algorithm and basic condition.

## Function Pointer Adaptors

- Function pointers are not directly composable
  - ◆ They are not proper objects in the sense that they can be constructed, and they are not adaptable
- Therefore adaptation is required when a function object type is expected

```
bool frequent(const counter_map::value_type &);  
...  
counter_map::iterator found =  
    std::find_if(  
        words.begin(), words.end(),  
        std::not1(std::ptr_fun(frequent)) );
```

For many uses the distinction between function pointers and function objects is transparent. However, when a function object type is specifically required, an unadorned function pointer will not work. For instance, where an adaptable function is needed or where an associative container requires a type for its ordering. Function adaptors let plain function pointers masquerade as adaptable function objects.

## Function Objects in Future

- Drawn from TR1 and based on Boost, C++11 offers developers simpler function object usage
  - ♦ The generalised *bind* family of function templates supports binding arguments to function calls in a simple way, superseding the C++98 binding facilities
  - ♦ The *function* polymorphic function wrapper is, in effect, a generalised function pointer that can hold arbitrary function pointers and function objects — it is often used with *bind*

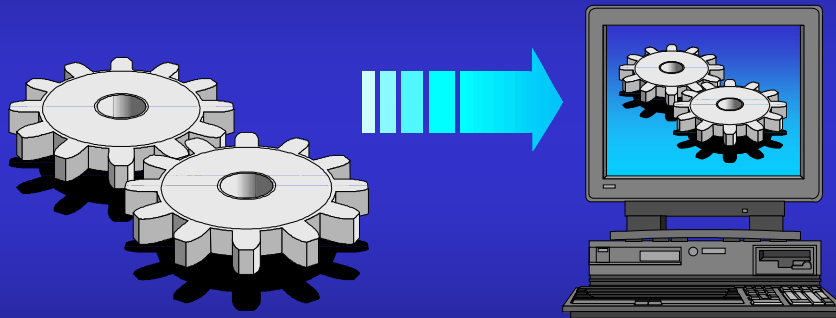
The generalised `bind` supports binding arguments to function calls: it yields a function object that calls the target function, member function or function object, passing explicit and prebound arguments to it. This facility encourages and supports a more convenient and functional programming style than is possible with existing or custom binders. Indeed, `bind` effectively displaces many of the "sort of" binding facilities in the standard — `std::bind1st`, `std::bind2nd`, `std::mem_fun`, etc.

Function pointers can point to plain functions, but not function objects or member functions. It is difficult to write generalised callback code without wrapping, introducing virtual functions or templating. Another issue is that the result type of a `bind` is unspecified — only its capabilities are specified — so a corresponding variable cannot be declared. The `function` class template plays the role of a generalised function pointer.

## Summary

- Function objects extend the basic capabilities of functions used for callbacks
  - ◆ Statefulness
  - ◆ Efficiency
  - ◆ Composability and adaptability
  - ◆ Generalised calling
- The standard classifies function object concepts and provides a core set of features

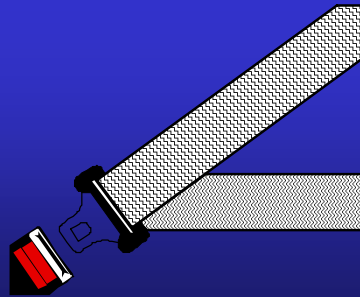
# Exception Safety



*C++ Advanced*

# Exception Safety

- Objectives
  - ◆ Outline concepts and techniques for clear and safe exception handling
- Contents
  - ◆ Exception objects
  - ◆ Standard exceptions
  - ◆ Handling exceptions
  - ◆ Levels of exception safety
  - ◆ Approaches to exception safety



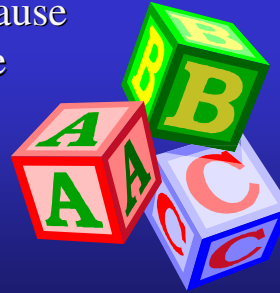
## *try and catch*

- To handle an exception, a *try* block must enclose code that may throw an exception
  - ♦ The *throw* may be directly within the block or within a function ultimately called from the block
- A *catch* handler is associated with the *try*
  - ♦ It specifies an exception type to catch
  - ♦ Call stack unwinds until a matching *catch* is found
  - ♦ The thrown object is normally caught by reference
- Execution resumes following the *catch*

The code that is to be attempted is placed within a `try` block. An exception arises as a result of a `throw` expression, which may be either explicit within the `try` block or present in the call chain from within it. The call chain is unwound until a `catch` – associated with a `try` – matches the exception type (either directly or as its base). The `catch` block is executed and the exception is considered handled.

## Exceptions Are Modular

- Discontinuous control flow generally presents an obstacle to simple refactoring
  - ♦ How do you *break* across a method-call boundary? Or *return* across two method calls? Or even *goto*?
- Exceptions are different because they preserve block structure
  - ♦ They are invariant with respect to refactoring



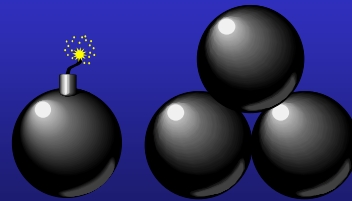
Error return codes tend to encourage a verbose coding style where most of the code is structured to propagate the error. Error code propagation increases the number of explicit paths in your code, which by definition makes the code more complex. Return values should, on the whole, be used to return useful values rather than good-news–bad-news bulletins.

Hang on, don't exceptions contradict the message about smooth flow, jumpy code, salmon, streams, etc? Not at all. One of the main problems with jumpy code is with respect to — or to be precise, its disrespect for — the modularity of block structured code. Refactoring one large method into many is often a process of splitting out a specific sequence or loop, giving it a name and figuring out what local variables need to be passed in and results returned. Data flow respects modularity, but what do you do with control that doesn't flow? How to make the effect of a `break` or early `return` statement non-local is less obvious than just passing and returning copies of local variables. Often extra status values have to be introduced and passed around in a game of pass the control flow. Exception flow, however, is trivial to refactor as it remains unchanged: an exception leaves a block and an operation, or an operation called by an operation, in the same way.



# Exception Objects

- Exceptions are referentially transparent
  - ♦ Informational or stateless objects
  - ♦ Typically live in a class hierarchy
- Defining a class for exception objects...
  - ♦ Catch by reference rather than by copy
  - ♦ Support copy construction
  - ♦ Typically should not support assignment
  - ♦ Single argument constructors should be *explicit*



© Curbralan Ltd

Exception Safety 201

C++'s exception mechanism affords transfer of both control and information from one point of execution to an earlier point in the event of bad news. There is a strong temptation to resort to built-in or conventional value types, e.g., throwing a `const char *` or `string`. Such use is clearly permitted by C++, but is considered a poor practice: conventional data types used for a different, unrelated purpose; within the program their presence communicates little except, perhaps, "there was an exception"!

Idiomatically an exception object should satisfy the following:

- Exceptions should be class types written especially for the purpose.
- Exceptions often live in a class hierarchy that is also a type hierarchy: The classification of exceptions is captured by public inheritance. For uniformity, the class hierarchy is often based on an existing hierarchy, e.g. the one rooted in the `std::exception` class. It is recommended that such hierarchies have concrete leaves, i.e. excluding the leaf classes all the classes are abstract: The standard hierarchy does not follow this.
- C++ requires exceptions to be copyable at the point of throwing, therefore they need to support copy construction.
- However, support for the assignment operator is not recommended: exceptions are thrown and caught, whereas assignment is for conventional values; assignment for objects in a class hierarchy can be problematic both pragmatically and philosophically.
- Exceptions may or may not have state. Sometimes the type of the exception is sufficient information.

## *std::exception*

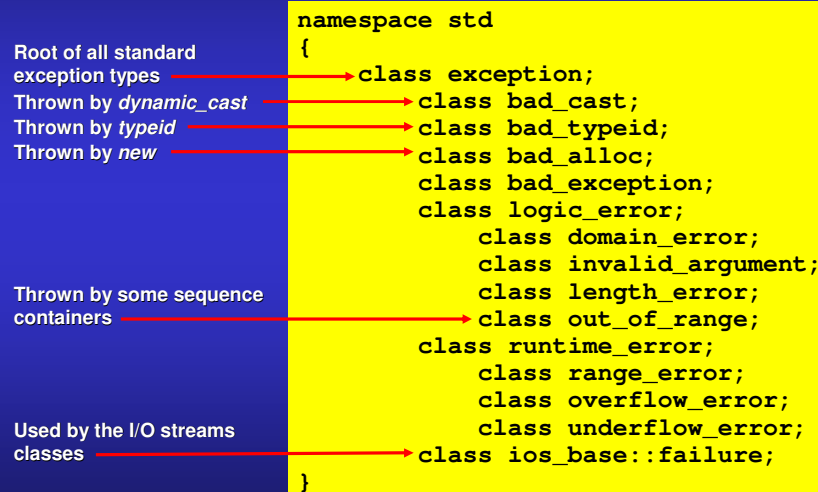
- The root exception class is provided as part of the language support library
  - ♦ Should also use as base for application exceptions

```
namespace std
{
    class exception
    {
    public:
        exception() throw();
        exception(const exception &) throw();
        virtual ~exception() throw();
        exception & operator=(const exception &) throw();
        virtual const char * what() const throw();
        ...
    };
}
```

Standard C++ provides a standard exception class, `std::exception`. This is used as the base for standard library exceptions, e.g., `std::out_of_range`, but it may also be used as the base for user defined exceptions.

Note that, perhaps questionably, `std::exception` is a concrete class at the root of a hierarchy, and also supports copy assignment. Neither feature makes that much sense when considered more deeply.

# The Standard Exception Hierarchy



Some of the standard exceptions are part of the library's core language support. This includes `std::exception` itself, as well as `std::bad_cast`, `std::bad_typeid` and `std::bad_alloc`. The `ios_base::failure` exception is intended solely for the I/O streams library — it does not attempt to integrate with the remainder of the exception hierarchy.

All the other standard exceptions are focused on classifying domain container-related or numeric precondition and postcondition violations. Only a few are actually used by the standard library.

# Handling Different Exceptions

- Multiple *catch* handlers may be associated with a *try* block
  - ♦ At most one will be matched and executed

```
try
{
    ... // normal code to be executed
}
catch(no_data &)
{
    std::cerr << "no data" << std::endl;
}
catch(failure & caught)
{
    std::cerr << caught.what() << std::endl;
}
```

When an exception is thrown, `catch` clauses are matched in turn in the order of definition. When a match is found, the `catch` clause is executed and the exception is considered handled.

## Catching Exceptions in a Hierarchy

- It is possible to catch an exception object with respect to its base class
  - ♦ Hence exceptions should be caught by reference

catch handlers  
are tried in the  
order they  
appear

```
try
{
    ...
}
catch(std::bad_alloc &)
{
    std::cerr << "out of memory" << std::endl;
}
catch(std::exception & caught)
{
    std::cerr << caught.what() << std::endl;
}
```

It is possible to catch exceptions at a general level of interest. It is also possible to extend the types of exceptions available without causing problems for existing code.

## Exceptions Should Be Layered

- Exception types should not reduce encapsulation of different layers
  - ♦ An exception may be caught and then re-thrown as a different type

```
...  
try  
{  
    ...  
}  
catch(raw_exception & caught) ← Catch exception type  
{                                that should remain  
    throw translated_exception(); ← Rethrow as exception  
}                                type that is visible  
                                outside subsystem
```

Exceptions can be caught only to be rethrown again. The most common use of this mechanism is to preserve encapsulation of given subsystems. Implementation-related exceptions should not stray outside a subsystem as types that are not supposed to be known or relied upon outside that subsystem. They should be rethrown as exceptions that make sense to the caller, while preserving the encapsulation of the subsystem by reducing type leakage.

## Refactoring Duplicate *catch* Code

```
try
{
    ....
}
catch(...)
{
    handle_exception();
}
```

The ability to rethrow exceptions means that common exception handling can be refactored into a common function.

```
void handle_exception()
{
    try
    {
        .... // common code for all handlers
        throw; // rethrow
    }
    catch(const application_exception &caught)
    {
        .... // handle application exceptions
    }
    catch(const std::exception & caught)
    {
        .... // handle other known exceptions
    }
    catch(...)
    {
        .... // handle unknown exceptions
    }
}
```

It is often the case that separate `catch` handlers have common code. However, there is no way to merge the `catch` clauses directly to factor out such code. Similarly, many `try` blocks in different functions share common tail end of `catch` handlers: they may have their own specific handlers and then other cases that are common.

The ability to rethrow a caught exception from within a function called from within a `catch` block offers a means of modularising and factoring out such common code, making the code both clearer and less affected by change.

# Plug-in Exception Handling

```
void execute(void action(), void handler())  
{  
    try  
    {  
        action();  
    }  
    catch(...)  
    {  
        handler();  
    }  
}
```

An operation that might throw an exception of some type not known to *execute*

A handler callback called in the event of an exception from *action*

```
void std_exception_handler()  
{  
    try  
    {  
        throw;  
    }  
    catch(std::exception & caught)  
    {  
        ...  
    }  
}
```

An example of a callback handler for exceptions

Rethrow the current exception

Catch specific exception type of interest

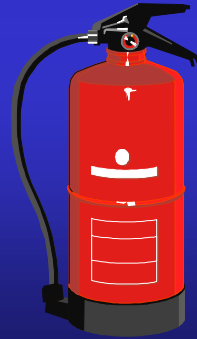
Exception handling sometimes appears to resist refactoring of duplicate code or the classic pluggability of callbacks. However, a simple technique supports exception-handling callbacks that can be used to parameterize a general framework with specific exception types. If a callback is called in a `catch` block the current exception is still active and can, in principle, be rethrown using a plain `throw`. Therefore, the convention that the callback author must follow is to rethrow and then catch the specific types of interest. This isolates the general code from the specific types.



## Levels of Exception Safety

- Five levels of exception-safety guarantee...

- ♦ *No guarantee* offers no guarantees
- ♦ *Leak guarantee* allows stable destruction in the face of leaked resources
- ♦ *Weak guarantee* ensures that throwing object will not leak or corrupt resources
- ♦ *Strong guarantee* ensures that the throwing object will remain as before
- ♦ *No-throw guarantee* ensures that exceptions are never thrown



The presence of exceptions in a program can have a significant impact. It is possible, if care is not taken, that an object can be left in an indeterminate state and therefore, by implication, the whole program will be in a potentially unstable state.

Exception safety is the stability of code in the presence of exceptions. Safety – and hence stability – can be classed at one of five levels:

- The *no guarantee* states that in the presence of exceptions anything can happen, i.e., there is no guarantee of sensible and stable behaviour.
- The *leak guarantee* states that in the presence of exceptions, the throwing object may leak resources. There is no guarantee that the object will be in a usable state, but the object will be safely destructible.
- The *weak guarantee* (a.k.a. the *basic guarantee*) states that in the presence of exceptions, the throwing object will not leak resources. The object will be in a stable and usable, albeit not necessarily predictable, state.
- The *strong guarantee* states that the program's state will remain unchanged in the presence of exceptions, that is, commit-or-rollback semantics.
- The *no-throw guarantee* promises that exceptions are never thrown.

## Achieving Exception Safety

- Developing with exceptions is a very different context to developing without
  - ♦ Existing code may be unsafe
- Three basic approaches to exception safety...
  - ♦ Exception safety through exception awareness and explicit handling
  - ♦ Exception-neutral code
  - ♦ Exception-free code



Exception safety can be achieved in a number of ways:

- Code may be exception unaware, which means that it will be prone in the face of error.
- Code may exception aware, scaffolded explicitly with `try`, `catch` and `throw` to ensure that restabilising action is taken in the event of a thrown exception.
- Exception neutral code is code that works in the presence of exceptions, but does not require any explicit exception handling apparatus to do so.
- Avoiding the use of exceptions in certain contexts.

With respect to the last point, we might wish to consider destructors. For automatic storage objects the end of an object's life can come about naturally or because the stack is winding as a result of an thrown exception. What if the destructor itself throws an exception? For normal control flow an exception will be propagated as expected; when there is already an exception already in progress the program will call `terminate`. There may be at most one exception active per thread of control; a second cannot be handled by the runtime, and so `terminate` is called to bring the program to a premature end.

Even without this serious side effect, it is not recommended that destructors throw exceptions. If an exception is thrown there is little that the caller would be able to do about it: the object would have been destroyed and so there would be no object on which to perform a recovery or retry attempt. It is also worth noting that the standard library provides no guarantees of working in the presence of exception destructors.

# Handle–Body Objects

- Classes may be split
  - ♦ An outer handle part
  - ♦ An inner body part containing the representation
- Motivation...
  - ♦ Information hiding
  - ♦ Exception safety
  - ♦ Smaller stack footprint
  - ♦ Inheritance-based variation
  - ♦ Representation sharing

```
class handle
{
public:
    ...
private:
    class body;
    body * self;
};
```

```
class handle::body
{
public:
    ...
};
```

There are many motivations for a Handle–Body object configuration. It is particularly suited to concrete classes, such as value object classes, as opposed to decoupling through a base interface class of only pure `virtual` functions.

For example, the Cheshire Cat idiom (also known as the Fully Insulating Concrete Class idiom and the Pimpl idiom) is perhaps one of the oldest class restructuring idioms in C++, dating back to Glockenspiel's `CommonView` library in the late 1980s. Thanks to nested forward declarations, modern C++ provides more encapsulated support for the idiom than early C++.

The whole representation of an object has been removed from the header file, leaving only a trace — no more than a smile — in the form of a pointer. The type of the pointer is fully elaborated in the corresponding source file that defines all the member functions.

# Unsafe Handle–Body Assignment

```
class handle
{
public:
    handle &operator=(const handle &);
    ...
private:
    class body;
    body * self;
};
```

*The classic canonical C++ assignment form is not exception safe, although it is clearly self-assignment safe*

```
handle & handle::operator=(const handle & rhs)
{
    if(this != &rhs)
    {
        delete self;
        self = new body(*rhs.self);
    }
    return *this;
}
```



© Curbralan Ltd

Exception Safety 212

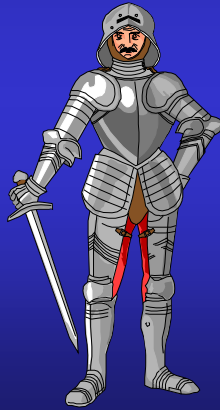
When an exception is thrown it is important to leave the object from which it was thrown in a consistent and stable state: the changes should be rolled back; the object may be left in a partial state that is still meaningful; or, the object is cleared out and marked as being in a detectable bad state. The consequences of not doing this can result in unsafe or unstable applications – unfortunate since the aim of introducing exception handling is to achieve quite the opposite!

The example above shows a seemingly harmless assignment operator. An initial inspection suggests that it follows all the good guidelines for writing an assignment operator: there is a check to prevent corruption in the event of self assignment; the previous state is deallocated; new state is allocated; `*this` is returned as in any normal assignment operator. However, what if the constructor of the `body` class were to throw an exception? Or if `new` were to fail and throw `bad_alloc`? Here be dragons!

A failed creation will result in an exception and control will leave the function, leaving `self` pointing to a deleted object. This stale pointer cannot then be safely dereferenced, with the result that the object is unstable and unsafe. When the `handle` object itself is destroyed, e.g. automatically at the end of a function, the attempt to `delete self` will likely wreak havoc in the application.

# Explicit Exception Handling

*This code is safe...  
but complex*



© Curbralan Ltd

```
handle &handle::operator=(const handle & rhs)
{
    if(this != &rhs)
    {
        body * old_self = self;
        try
        {
            self = new body(*rhs.self);
            delete old_self;
        }
        catch(...)
        {
            self = old_self;
            throw;
        }
    }
    return *this;
}
```

Exception Safety 213

It is initially tempting to construct a `try catch` block to guard against exception instability. This can sometimes lead to quite elaborate and intricate code; for some functions the solution sometimes looks worse than problem!

Although necessary in a few cases, constructing exception-handling scaffolding to ensure exception safety is not the simplest answer; exception-neutral code, as we shall see, typically offers simpler, more elegant, and more efficient solutions. The Copy Before Release pattern illustrates an exception-neutral alternative, and there is an equivalent approach based on the provision of a non-throwing `swap` function.

However, explicit handling for safety is still a strongly tempting path to follow. The problem is that such a solution style attracts a way of thinking about the problem that is too close to the problem; a piecemeal solution is grown by grafting onto the original exception unsafe code. If you want to reach exception safety, reflect that setting out from exception unsafety is probably not the best starting point!

# The Copy Before Release Idiom

- A far simpler exception-safety solution through careful ordering of actions
  - ♦ State passes through stable intermediate forms

```
handle & handle::operator=(const handle & rhs)
{
    body * old_self = self;
    self = new body(*rhs.self);
    delete old_self;
    return *this;
}
```



*There is no requirement to perform a self-assignment check as the ordering is also self-assignment safe*

## Context

- A class has been implemented as handle/body pair in C++.
- The body is safely copyable – type shallow (ordinary copy constructor) or deep (Cloning) as appropriate.

## Problem

- Ensuring update by copy – such as assignment – is exception safe.

## Forces

- Any of the steps taken in performing the update may fail, resulting in a thrown exception. Partial completion of the steps may leave the handle in an unstable state.
- The result of update, successful or otherwise, must result in a stable handle.
- Self assignment must also result in a stable handle.
- After successful completion of the assignment the handle on the left hand side of the assignment must be behaviourally equivalent to the one on the right.
- Update, successful or otherwise, must be non-lossy, i.e. no memory leaks.

## Solution

- Alias the existing body before taking the body copy.
- Perform the body copy and bind to the handle before releasing the existing body via the alias.

## Consequences

- The existing body is not deleted before the body copy has been attempted. Therefore, a failed body copy will not result in an unstable handle.
- Failed body release may result in resource loss, but the assignment will have succeeded and have left the handle in a stable state.
- The ordering accommodates safe self assignment at the cost of a redundant copy.
- If the body copy preserves behaviour equivalence, a successful assignment will preserve it for the composite handle/body object.
- The solution can be used in conjunction with the schema for copy assignment from the Orthodox Canonical Class Form.

# The Non-throwing Swap Idiom

```
class handle
{
public:
    handle(const handle &);
    void swap(handle & other)
    {
        std::swap(self, other.self);
    }
    handle & operator=(const handle & rhs)
    {
        handle copy(rhs);
        swap(copy);
        return *this;
    }
    ...
private:
    class body;
    body * self;
};
```

*There is no requirement to perform a self-assignment check or any explicit memory management*

The presence of a non-throwing `swap` member function, for exchanging the representation of one object with another of the same type, also offers a route to strong exception safety.

The assignment operator takes a copy of the right-hand side. This may throw an exception, but if it does the current object is unaffected. The representation of the current object and the copy are exchanged. The assigned object now has its new state and the copy now holds its previous state, which is cleaned up on destruction at the end of the function.

The control flow in this idiom is the same as that of Copy Before Release, although achieved through an intermediate object rather than directly.

## Acquisition and Release

- Resource acquisition and release calls are commonly paired around usage sequences
  - ♦ E.g., *open* a file, use it, *close* it
  - ♦ E.g., *lock* an object against multithreaded access, use it, *unlock* it
- There is a need to...
  - ♦ Abstract this commonality
  - ♦ Ensure exception safety



Most object abstractions are largely data or function based, rather than attempting to abstract control structure. The flow schema

```
acquire resource
use resource
release resource
```

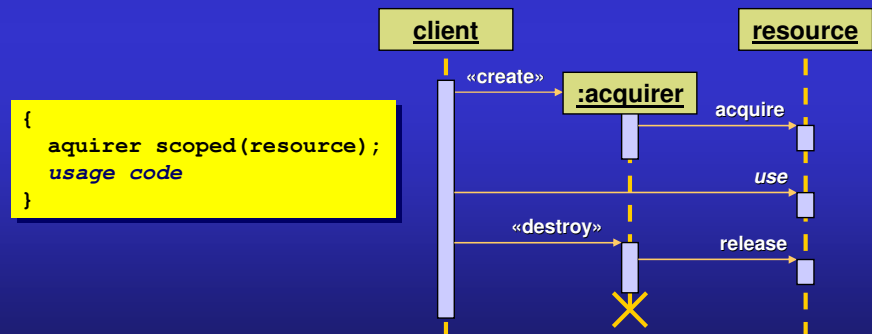
is sufficiently common that of itself it might be considered a programming "turn of phrase". For instance it is used when dealing with record locking and unlocking, with opening and closing a file within the same block, and setting and resetting a wait cursor around a task.

Such repetition suggests some missing abstraction, and is both error prone (from the programming perspective) and prone in the presence of error (not exception safe). What is needed is some way to capture the control flow schema and encapsulate it so it is simple and exception safe.



## The Execute-Around Object Idiom

- Places control with a helper object
  - ♦ Lifetime of helper encloses usage
  - ♦ Also known as *resource acquisition is initialization*



© Curbralan Ltd

Exception Safety 217

In C++ a constructor is called on creation of an object for the sole purpose of initialising it, i.e. it describes the "boot sequence" for an object. Conversely a destructor is automatically called at the end of an object's life to finalise or clean it up, i.e. to shut it down in an orderly fashion. Importantly, in C++ the calling of a destructor is deterministic: the life of a local stack variable is tied to its enclosing scope; the life of an object allocated dynamically using the `new` operator comes to an end by an explicit call to `delete`.

The former property, that of tying lifetime to scope, is what underpins the Execute-Around Object idiom. A helper object is declared with a reference to the target object whose member functions must be paired around use. The helper object calls back in to the target object on creation to acquire or initialise the resource. In its destructor it automatically calls back to release or finalise the target's resource.

Sometimes the initialiser action is performed before the constructor of the helper is executed; sometimes no initialiser action is required. In truth, the essence of this pattern is in the execution of the destructor and not the constructor, although this is clearly common. The relationship between helper object lifetime and the destructor is crucial, as this means this idiom addresses not only control flow abstraction, but also exception safety.

Such helper objects are good examples of the fine grained helper objects that can simplify an overall implementation, as opposed to the coarse grain abstractions apparent in the business model or user interface of a system.

## The Smart Pointer Idiom

- A C++-specific adaptation of the Proxy pattern
  - ♦ Classes that support pointer-like protocol
- Different kinds of 'smartness' are possible
  - ♦ From memory management to synchronisation

```
template<typename ElementType>
class smart_ptr
{
public:
    ElementType & operator*() const;
    ElementType * operator->() const;
    ...
};
```

Pointers conventionally represent a level of indirection to a target; structurally this is a feature shared with Proxy classes. In C++ the Smart Pointer idiom builds on the understood protocol for pointers, specialising the Proxy pattern for certain examples where access control takes the same form for all operations.

The essential features of a Smart Pointer are its dereferencing operations: `operator*` and `operator->` are overloaded. Both of these operators provide access to the indirectly referenced object, but do not themselves modify the proxy; hence they are `const`. The `operator*` typically returns a reference rather than a value, and the `operator->` must return something that in turn supports an `operator->`, such as a raw pointer.

## *std::auto\_ptr*

```
template<typename ElementType>
class auto_ptr
{
public:
    typedef ElementType element_type;
    explicit auto_ptr(element_type * = 0) throw();
    ~auto_ptr() throw();

    element_type & operator*() const throw();
    element_type * operator->() const throw();
    element_type * get() const throw();
    ... // other members related to transfer of custody
private:
    element_type * ptr;
};
```

*NB: Does not qualify as either CopyConstructible or Assignable.*

The Execute-Around Object idiom is found at the heart of the misnamed Resource Acquisition is Initialization idiom; misnamed because the essence of what makes this pattern work is the destructor. In many cases, resource acquisition occurs independently of the Execute-Around Object, as in the case of memory acquisition. Perhaps *Resource Release is Finalization* would be a better name for the resource based applications of Execute-Around Object.

It is normally the responsibility of an object user to deal explicitly with the ownership and lifetime of objects. The standard `auto_ptr` template simplifies particular cases: heap object whose lifetime should be bound safely to an enclosing scope; exception safe transfer of object ownership through arguments or return values.

The design of the standard `auto_ptr` has evolved through many unsatisfactory compromises to the point that some of its semantics and its full interface are less than intuitive. However, it is standard and the `const auto_ptr` idiom allows it to be used simply and in a manner closer to its original intended purpose, i.e. with no transfer of ownership.

## Summarising the Transactional Style

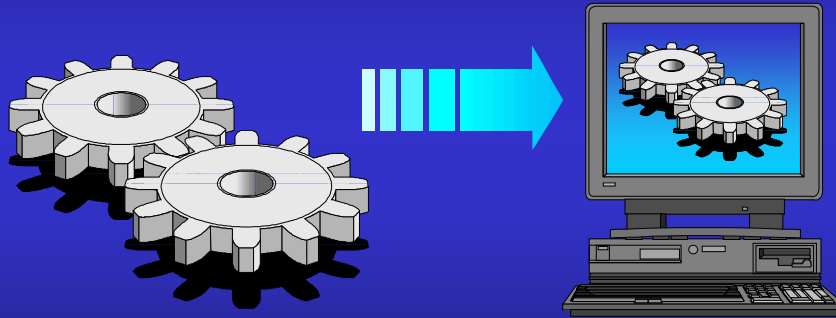
- The simplest approach to achieving exception-safe state change is...
  - ◆ Ensure that a stable return can be returned to, e.g., by saving state
  - ◆ Prepare the change
  - ◆ Commit the change, e.g., by using a non-throwing action to enact the state change
  - ◆ Clean up, e.g., release the previous state, which may be automated by a destructor

Many techniques for exception safety follow a common underlying model based on a transactional style of coding. For instance, perform all exception-fallible actions above a particular point in the code where a commit action of some kind occurs and ensure that the commit action satisfies the no-throw guarantee. Destructors can complete or clean up the remainder of the action.

## Summary

- Exception safety describes the stability and reliability of code in the presence of exceptions
  - ◆ There are different levels of guarantee
- There are a number of approaches to achieving the different levels of exception safety
  - ◆ Exception neutral code is cleaner and clearer than exception-aware code
- Encapsulation is often the key to safety

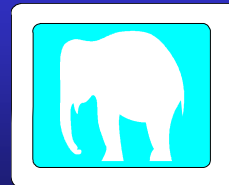
# Dynamic Resource Management



*C++ Advanced*

# Dynamic Resource Management

- Objectives
  - ◆ Outline a number of techniques for safely managing resources, especially memory
- Contents
  - ◆ Resource management
  - ◆ Encapsulated memory management
  - ◆ Dynamic memory guidelines
  - ◆ Object factories
  - ◆ Reference counting



## Resource Management

- Objects that can become scarce and whose scarcity would be problematic are *resources*
  - ♦ This includes dynamically allocated memory, file handles, synchronisation primitives, connections and any object that directly contains one of these
- Resource management should ideally be explicitly balanced or fully encapsulated
  - ♦ In C++ the resource that most obviously causes trouble is dynamic memory

Programs typically involve a number of resources to achieve their goals, whether it is something like memory for objects on the heap or connections to files or databases. This makes timely and correct resource management the concern of the programmer.



## Private Memory Management

- Memory management should be encapsulated as much as possible
  - ◆ Users of a type should not be fully responsible for doing object book keeping of the instances
  - ◆ Value objects are an example of types that should not expose their memory management
- Programmers are often tempted to include "helpful" extras to allow control of memory
  - ◆ These are rarely helpful, and always confusing

Memory management is one of the housekeeping tasks that can run out of control in a C++ program if a few simple guidelines are not followed. Encapsulation of memory management is the simplest guideline. Most attempts at making memory management flexible are misguided and reduce the robustness of code. Worse, programmers who are not aware that memory management is a design consideration rather than an incidental detail tend to leave ad hoc memory management schemes in their wake.

## C-style Collection Management

- Classes manage multiple objects manually
  - ♦ C-style arrays
  - ♦ Hand-rolled lists
- Classes have been, and are still being, written this way

```
class dictionary
{
public:
    size_t size() const;
    const std::string * lookup(
        const std::string &) const;
    bool insert(
        const std::string & key,
        const std::string & value);
    ...
private:
    std::string * keys;
    std::string * values;
    size_t      used;
    size_t      reserved;
};
```

Legacy code often still treats C++ as simply a better C, exposing a lot of low-level memory management details in higher-level classes. The most common example of this is the use of `new [ ]` arrays paired with a sizing integer. Another example is specifically hand-rolled linked lists or hash tables. This makes the owning class less cohesive as it is focused on two very different tasks: (1) its principal purpose and (2) its housekeeping detail for managing collections of data.

This approach makes code verbose, repetitive, error prone and less encapsulated. It is also not restricted to legacy code: many new programmers, especially with a C background (or those taught C++ with a C bias), use this low-level approach.

## Using Container Members

- The standard library provides a well-defined set of container template classes
  - ♦ Adequate for most needs
  - ♦ Easy to define new container types based on the standard container model

```
class dictionary
{
public:
    size_t size() const;
    const std::string * lookup(
        const std::string &) const;
    bool insert(
        const std::string & key,
        const std::string & value);
    ...
private:
    std::map<
        std::string,
        std::string> entries;
};
```

A variant of introducing a specific type is the idea of introduce a container member. This transforms data members focused on collection management into one of the standard container types, e.g., `vector` or `map`.

## Explicit Memory Management

- Where memory management is explicit, it is best expressed and used symmetrically
  - ◆ Ownership is clearest, bugs are less likely
- Where possible, the same scope should be responsible for both acquisition and release
  - ◆ E.g., balanced factory and disposal methods
  - ◆ E.g., preferring hierarchical ownership to memory-managing smart pointers

Many problems with C and C++ memory management are not related to the nature of manual management, but the unclear design aims of different parts of a system. A common problem is that the rules of object ownership are unclear, often because they are asymmetric.

A common salve in C++ is to enrol smart pointers to take the strain of memory management design. Memory-managing smart pointers can be very effective in addressing certain problems, but they are often overused when they are used. They should not add dependencies into interfaces where a plain pointer would be both adequate and general enough. They should also not be used as an alternative to clear ownership models. There are many examples where memory managing smart pointers are used when fully encapsulated and hierarchical ownership models would be simpler and more appropriate. Such smart pointers are most suitable for cases where simpler memory management models are insufficient, and should be used judiciously rather than liberally.

## Dynamic Memory Safety Guidelines

- Use the matching deallocator function
  - ◆ I.e., use *delete* with *new*, *delete[]* with *new[]*, *free* with *malloc*
- Make sure that an allocated object is deleted precisely once
  - ◆ Multiple deletion of an object leads to undefined behaviour
  - ◆ Forgetting to delete an object leads to leaks
  - ◆ But it's OK to delete null



Even if the same underlying heap is used, it is not safe to mix `new` and `delete[]` or `malloc` and `delete`.

## Overloading *new* and *delete*

- It is possible to redefine *new* and *delete* either globally or for a class
  - ♦ Sometimes useful as an optimisation; most often useful for tracking memory usage

```
class some_type
{
public:
    static void operator * new(std::size_t size);
    static void operator delete(void * to_delete);
    ...
};
```

The signatures of `operator new` and `operator delete` are essentially the signatures of `malloc` and `free`. There is an additional variant of `operator delete` that applies only to class-based versions, and this receives a second argument indicating the size of the returned memory. Note that because `new` and `delete` can only be static, the use of the `static` keyword is optional.

## Lifetime Issues and Encapsulation

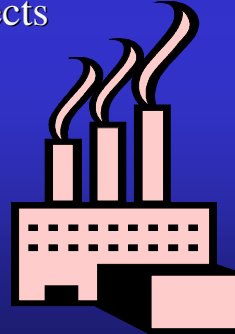
- Not all object creation and deletion can be reduced to *new* and *delete*
  - ◆ Sometimes concrete object details should be hidden
- In C++ object lifetimes are deterministic
  - ◆ Heap objects require either...
    - An explicit policy for ownership...
    - Or a way of tracking when they are no longer referenced
  - ◆ Object custody should always be clear in a design
    - This is where bugs commonly occur

At its simplest, on-the-fly object creation is performed using `new` and deletion using `delete`. There are times when the details of the concrete object created — including its type and any other management of its creation — needs to be hidden from view in order to reduce dependencies in a program.

It should be possible, when looking at any pointer in a C++ program, to determine which object or scope is responsible for its management. Unclear or confusing object custody tactics help to cause bugs. Policies must either be clearly with respect to a manager or boss object, or managed through a set of intermediate objects, such as smart pointers.

# Object Factories

- Knowledge for creation of an object can be delegated and encapsulated
  - ♦ Exact detail of creation is hidden from the caller
- Object factories creates other objects
  - ♦ Being an object factory may be the responsibility of a single operation, a whole object, or a class



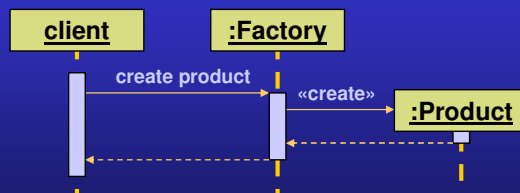
The separation of interface from implementation can lead to creational issues. If the client that uses an object through an interface is also aware of its concrete type and is the party responsible for creating it, there is no issue. However, if the client is only aware of the usage interface rather than the concrete creation type, but is also responsible for instigating the act of object creation, there are apparently opposing design forces that need to be resolved.

There are many patterns that address creation issues by encapsulating creation knowledge in some way. Objects that are responsible for creating other objects are known, for fairly obvious reasons, as factories.



# The Factory Method Pattern

- Problem...
  - ◆ How can you create an object without knowing its concrete class?
- Solution...
  - ◆ Provide a method for creation at the interface level of an object that performs actual creation



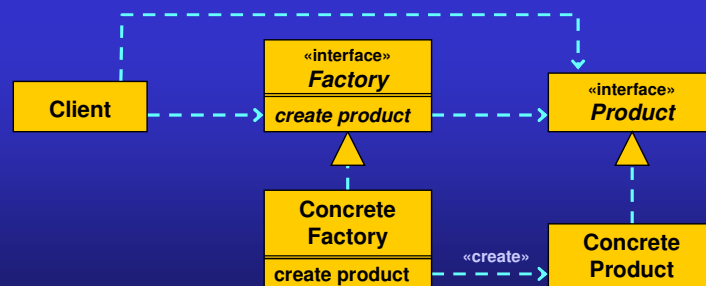
Parallel hierarchies are often created using the Factory Method pattern. The intent for this pattern is summarised as

*Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.*

Thus, the idea of creation is contracted out at the interface level, but its realisation is deferred to one or more subclasses.

# Factory Configuration

- Mirroring the hierarchy of what is created...
  - ♦ An interface for creation is provided
  - ♦ The responsibility for creation is deferred to an implementing subclass



Generically we can see that in the two hierarchies, one is the product hierarchy:

- A product interface defines how to use a created object. In the purest case this will be an interface, but it may alternatively be an abstract classes including some implementation.
- A concrete product class is the type instantiated against the interface.

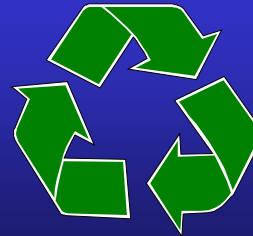
And the other is the creator or factory hierarchy:

- The creator interface defines how to create instances that implement the product interface. As with the product, this may be either an interface or an abstract class.
- The concrete creator class supports the creator interface and implement the creation of concrete product objects.

# Disposal Method

- Problems...
  - ♦ Who cleans up factory products after use?
  - ♦ How can you reduce dynamic memory usage?
- Solution...
  - ♦ Give the created object back to its creator

```
class factory
{
public:
    virtual product * create() = 0;
    virtual void dispose(product *) = 0;
    ...
};
```



Mirroring Factory Method, Disposal Method answers the question of who is responsible for the disposal of a factory created object by returning the object to the factory, thus localising all knowledge of creation/disposal in a single place. This is applicable both in garbage collection environments, such as Java, and explicit lifetime managed ones, such as C++.

Disposal Method may either be implemented either by giving the factory the object back directly, or by providing a method in the object that returns it to its creator.

Importantly, because Disposal Method hides the policy of the factory, objects can be cached and recycled to save on the cost of heap management. In CORBA systems there is no implicit lifetime management model, and so Disposal Method is applicable for many utility objects (which should also be *transient* as opposed to *persistent* objects).

## Reference Counting

- Maintain usage count of a representation object in order to manage its lifetime
  - ♦ When the count falls to zero, the target object can be deleted
  - ♦ Can also be used to support other constraints
- There are many different configurations that support the general pattern
  - ♦ Idiomatic with respect to C++ and with different space and performance tradeoffs



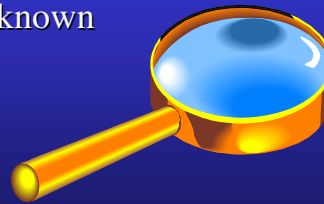
It is normally the responsibility of an object user to deal explicitly with the ownership and lifetime of objects. Reference counting puts the responsibility for management of object ownership into the association. The principle behind reference counting is to arrange to keep a shared usage count of the object pointed to, updating its value when they are shared or forgotten. On falling to zero the target object is effectively unused, and is deleted.

Reference counting can be used as a simplified form of garbage collection, as well as the basis for optimisation through representation sharing. Unlike garbage collection, which is typically implemented for other languages and some C++ systems as deferring collection to some time after an object has been forgotten, the cost of object collection for reference counting is normally paid as soon as the object is forgotten. This means that the timing of destruction is deterministic, which is necessary to avoid exhaustion for some resources, e.g. file handles. However, immediate collection is not a requirement of reference counting schemes: on falling to zero, counted objects can be placed in a morgue that is periodically cleared out.

Developers familiar with reference-counting schemes often know about reference counting in a particular context and for a specific task, i.e. they may be familiar with the string copy optimisation or with component object lifetime management, as expressed in COM through the `AddRef` and `Release` methods of the `IUnknown` universal base interface. In truth, there are many uses and different schemes for reference counting, as well as more caveats, than many know.

## Visibility of Reference Counting

- Hidden...
  - ♦ Separation of a single value based object into handle–body pair
  - ♦ Modification of shared representation leads to copy-on-write
- Explicit...
  - ♦ Use of a proxy to access a known shared object



© Curbralan Ltd

Dynamic Resource Management 237

Reference counting builds on the handle–body configuration of separating a concept into two parts:

- The body holds the content and data of interest and is the object whose usage is being counted. It is allocated dynamically on the heap.
- The handle provides and controls access to the body, as well as performing the book keeping associated with reference counting. It is a value based object that is normally a local variable or a data member; it is not normally a heap object.
- The reference counting process can be automated by controlling the construction, destruction and assignment of the controlling handle. Hence automated reference counting is exception safe. On falling to zero the target object is effectively unused, and is deleted.

A reference counted body may be classified as either *hidden* or as *explicit*:

- A hidden implementation is typically an optimisation of which the class user is unaware. A common example is to reduce copying for value based objects, such as strings, that might otherwise incur an unnecessary overhead. The solution is to share the representation until such a time as one handle needs to modify it, when it is then duplicated; this is the Copy on Write pattern.
- An explicit mechanism is used when the class user is aware of the counting facility on offer and is often responsible for the creation of the body. The handle is then typically a proxy, such as a smart pointer.

*Taligent's Guide to Designing Programs* [Taligent1994] offers a more detailed classification schema.

## Count and Body Configuration

- What is the physical relationship between the count and the body?
  - ♦ Count and body may be attached or detached
- How intrusive is reference counting with respect to the counted body?
  - ♦ Countability may be a property of the body or added and managed by the handle
- In some cases the count need not be held explicitly or may be managed externally

When adopting any kind of framework, the developer must be aware that in addition to the benefit they seek, there may be side effects such as an intrusive coupling from their classes to the classes of the framework.

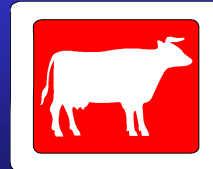
There are two basic approaches to reference counting objects:

- The count is part of the body object. The handle manipulates this part of the body object as necessary. The benefit is that only one object is being dealt with. The main disadvantage is that this is normally implemented to require the body class know not only about reference counting, but also about the particular reference counting mechanism, i.e. the body class is tied to a particular counting mechanism. This also means that if the object is not allocated directly on the heap, e.g. it is a local variable or it is a composite part of another object, the reference count is redundant overhead.
- The count is separate from the body object. The handle assumes nothing about the body type and uses a separate object to hold the count; the handle coordinates the existence of both the body and the count. The advantage is that any type – including built-in types – can be counted, and so the reference counting is non-intrusive. The main disadvantage is that two objects are required rather than one, and this has the potential to fragment and use up more memory than is strictly necessary, especially as one of the objects is (the count) is quite small; heap managers are not optimised for objects only a few bytes in size.

The examples used in the remainder of this section will explore these variations and what they mean for explicit reference counted body configurations; the techniques typically also apply to hidden reference counted bodies.

## Representation Sharing

- To reduce the cost of copying a body object when a whole handle–body is copied...
  - ♦ Share the body objects between copied handles
  - ♦ Reference count the body objects, so that they are deleted when no more handles point to them
  - ♦ Perform a copy-on-write (COW) when modifying
- However, there are COW issues...
  - ♦ Code complexity, thread safety and whether it is always an optimisation



The most common copy optimisation for handle–body objects is to share the representation of the object when a copy is made rather than make a deep copy that results in heap allocation. This means that copying is simple and cheap. Only when the actual body object is going to be modified does the 'real' copy occur to avoid aliasing surprises. This lazy, just-in-time model — commonly referred to as copy-on-write — defers the cost of allocation until the point it is absolutely needed. If it is never needed, the cost is not paid.

However, few things in life are for free: the sharing is not without overhead. For a start, it must be managed, which increases the complexity of the code. The referencing must also be tracked so that when — as a result of assignment or destruction — the body object is no longer referenced it is properly deallocated, and when only a single handle refers to the body, redundant deep copies are not made. There is also the question of thread-safe counting and change.

# Public Countability

```
template<class CountableType>
class countable_ptr
{
public:
    explicit countable_ptr(CountableType *);
    countable_ptr(const countable_ptr &);
    ~countable_ptr();
    countable_ptr & operator=(const countable_ptr &);
    ...
private:
    CountableType * body;
};
```

*The target body class must support a countable interface and define a count data member*

```
class CountableType
{
public:
    void acquire() const;
    void release() const;
    bool acquired() const;
    ...
private:
    ...
    mutable size_t count;
};
```

This approach requires that the body type support a countable interface, i.e. `acquire`, `release` and `acquired` functions. This means that the use of reference counting is intrusive and that the count is embedded explicitly within the body class; a base class can be used to factor out the common implementation. Note that the countable functions are `const` and the count data member is mutable: lifetime management is not part of a countable object's logical state, i.e. as in the rest of C++ lifetime management is independent of `cv` qualification.

The assignment operator for the `countable_ptr` class illustrates how the countable mechanisms are used in practice:

```
template<class Type>
countable_ptr<Type> &
countable_ptr<Type>::operator=(const countable_ptr & rhs)
{
    Type * old_body = body;
    body = rhs.body;
    if (body)
        body->acquire();
    if (old_body)
    {
        old_body->release();
        if (!old_body->acquired())
            delete old_body;
    }
    return *this
}
```



## Hidden Prefix Count

*countable\_ptr definition as before*

```
struct countable_new;  
extern const countable_new countable;  
  
void * operator new(size_t, const countable_new &);  
void operator delete(void *, const countable_new &);
```

*Overloaded operator new prefixes allocated body with counting header and countable\_ptr is written to use helper functions that manipulate this header*

```
struct countable_new_header { size_t count; };  
  
void acquire(const void *);  
void release(const void *);  
bool acquired(const void *);
```

*The target body type is independent of counting, but countable instances must be allocated using the overloaded operator new*

```
countable_ptr<type> ptr = new(countable) type;
```

What we would like to do is confer this capability on a per object rather than on a per class basis. Effectively we are after countability on any object, i.e., anything pointed to by a `void *`!

The first step is to realise that the `countable_ptr` class does not have to change externally. In fact it is possible to write a single `countable_ptr` such that its implementation need not change either and the same class template may be used for both the discreet and visible mechanisms. However, to keep things simple, consider that the first step is to assume that there exist acquisition and release functions that can perform the counting. `counted_ptr` may now be written in terms of these. The next step is to place the count ahead of the body in memory by providing an alternative version of operator `new`; the acquisition and release functions may now be written to manipulate this header.

Here is the modified assignment operator:

```
template<class Type>  
countable_ptr<Type> &  
countable_ptr<Type>::operator=(const countable_ptr & rhs)  
{  
    Type * old_body = body;  
    body = rhs.body;  
    acquire(body);  
    release(old_body);  
    if(!acquired(old_body))  
        delete old_body;  
    return *this  
}
```

## Associating Count

```
template<typename Type>
class counted_ptr
{
public:
    explicit counted_ptr(Type *);
    counted_ptr(const counted_ptr &);
    ~counted_ptr();
    counted_ptr & operator=(const counted_ptr &);
    ...
private:
    counted_link * link;
};
```

```
struct counted_link
{
    size_t count;
    Type * body;
};
```

*Target body type is independent of counting, and is two levels of indirection removed from the handle via the count*

```
class Type { ... };
```

Detached reference-counting mechanisms place the count outside the object being counted, such that they are handled together. The clear benefit of this is that this technique is completely unintrusive, with all of the intelligence and support apparatus in the smart pointer, and therefore can be used on classes created independently of the reference counted pointer. The main disadvantage is that frequent use of this can lead to a proliferation of small objects, i.e. the counter, being created on the heap.

The linked approach associates the handle with the count, and the count with the body. The principal benefit of these two levels of indirection is to allow the body to be replaced in some way, without affecting the handles. In this respect, additional properties and capabilities can be added transparently, such as dynamic load and unload on demand. A refinement to the code shown above that would reflect this would be to model `counted_link` as a class rather than a plain data structure.

## Independent Count

```
template<typename Type>
class counted_ptr
{
public:
    explicit counted_ptr(Type *);
    counted_ptr(const counted_ptr &);
    ~counted_ptr();
    counted_ptr & operator=(const counted_ptr &);
    ...
private:
    size_t * count;
    Type * body;
};
```

```
class Type { ... };
```

```
typedef ... size_t;
```

*Target body type is independent of counting,  
and is not physically related to the counter*

In common with the previous approach, this one keeps count and body separate. Now, however, they are completely separate, i.e., not even associated via a link. This reduces the number of levels of indirection, but means that the handle has a slightly larger footprint.

The logic for the assignment is as follows:

```
template<typename Type>
counted_ptr<Type> &
counted_ptr<Type>::operator=(const counted_ptr & rhs)
{
    Type * old_body = body;
    size_t * old_count = count;

    body = rhs.body;
    count = rhs.count;

    if (body)
        ++*count;
    if (old_body && --*count == 0)
    {
        delete old_count;
        delete old_body;
    }
    return *this
}
```

As with the other smart pointer classes, factoring out common implementation code into private member functions would make this simpler.

# Linked Handles

```
template<typename Type>
class counted_ptr
{
public:
    explicit counted_ptr(Type *);
    counted_ptr(const counted_ptr &);
    ~counted_ptr();
    counted_ptr & operator=(const counted_ptr &);
    ...
private:
    Type * body;
    counted_ptr * next, * previous;
};
```

```
class Type { ... };
```

*Target body type is independent of counting,  
and there is no explicit count*

Link handle objects together in a doubly-linked list and hold a pointer to the body object. When the links going to the previous and next handle are both null (or, in a circular configuration, pointing to `this`) the body is uniquely owned.

This style of reference accounting (it is not really reference counting because there is no explicit count) is most appropriate when there are operations that require traversal of all handles. Each handle will have a larger footprint than the other solutions considered so far, although only a single allocation is required per body.

## Reference Counting Issues

- Indexing copy-on-write objects...
  - ♦ Smart references
  - ♦ Invalidate sharing when indexed
  - ♦ Or simply use immutable objects
- Cycles...
  - ♦ Cyclic dependencies leading to orphaned rings of objects are possible with smart pointers
- Concurrency...
  - ♦ Thread safety of the count



Developers of reference-counted abstractions (and in some cases, unfortunately, also the users) must be aware that reference counting is not a panacea and comes with some caveats:

- Although officially classed as a query function, a function that grants access to part of the representation via a reference, e.g., `operator[]` on a string class, can lead to aliasing problems with a hidden shared representation. One solution is to prevent further sharing — thus avoiding any future problems, but also any further optimisation — another is to regulate the access by returning a smart reference. There is more discussion of the issues by Meyers in *More Effective C++*.
- The use of reference counting smart pointers will not work for objects with cyclic references, i.e. when an object refers to another object that refers directly or indirectly back to the first object, in both cases using a reference counted smart pointer. The cycle will keep all the objects alive with a reference count of at least 1, even if the main program has long since forgotten about them. In complex object models such situations can arise without programmers realising it, and it is something to be aware of.
- In the presence of multiple threads of control it is possible that smart pointers in two different threads may attempt to update the reference count simultaneously. Such a race condition can lead to subtle, and hard to reproduce errors (the worst of kind of error!). If the correct synchronisation strategy is not applied, the resulting performance may be disproportionately inefficient; this is particularly in the case of hidden counted bodies introduced as an optimisation!

## Reference Counting in C++11

- Drawn from TR1 and based on Boost, C++11 offers developers some standard smart pointers
  - ♦ The *shared\_ptr* template is templated and non-intrusive, and supports customised finalisation
  - ♦ The *weak\_ptr* template offers a safe, non-counting facility for pointing to *shared\_ptr*-managed objects
- There are still cases where a customised reference-counting solution is preferable
  - ♦ And, in some cases, not reference counting at all is an even simpler and more effective option

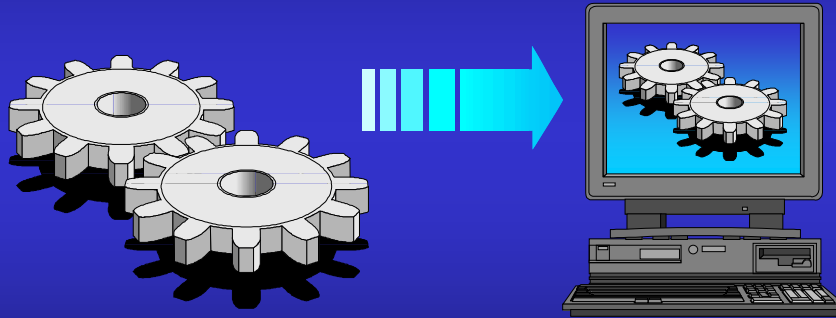
As already seen, a common advanced technique for managing shared-object lifetime is the use of reference counting. However, it is a chore that not all developers are comfortable to undertake. The `shared_ptr` class template addresses this common need. It is templated and non-intrusive. By default, `delete` is called on the object when the count falls to zero, but a custom deleter can be installed instead.

There is also need to have pointers to objects that do not influence the lifetime but are checkable: a `shared_ptr` affects lifetime and raw pointers cannot be checked for target object validity. The `weak_ptr` template addresses these issues. A `weak_ptr` can be initialised from a `shared_ptr`. It can be used to observe objects safely without modifying their lifetime. One use of this is to break cycles in reference chains.

## Summary

- Resource management is more than just a matter of *new* and *delete*
  - ♦ Creation and destruction can be encapsulated and strategies exist to keep things simple
- The timing of destruction can be simplified by using reference counting
  - ♦ Various idioms in C++ support the implementation of reference counting
  - ♦ Tracking references is not applicable to every situation

# Dependency Management

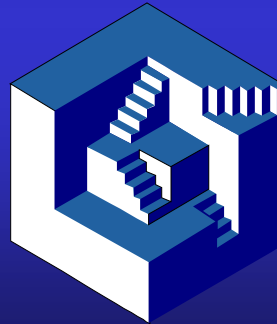


*C++ Advanced*



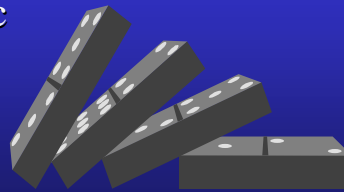
# Dependency Management

- Objectives
  - ◆ Outline the issues and solutions available in C++ for managing dependencies
- Contents
  - ◆ Header organisation
  - ◆ Forward declaration techniques
  - ◆ Dealing with inline functions
  - ◆ Interface classes
  - ◆ Generic decoupling



## Dependency Management Issues

- Large systems can become difficult to manage if the dependencies are not controlled
  - ♦ Comprehensibility, testability, extensibility, maintainability and buildability all suffer
  - ♦ The effect of change becomes severe
- Therefore, dependencies must be managed between classes, headers, etc
  - ♦ Refactoring may be needed to ensure this



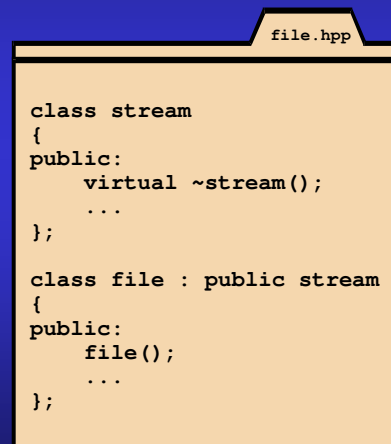
One of the features that typifies any architecture driven approach is the management of dependencies in a design. Dependencies should be managed throughout the runtime, design time and construction time of a system. In terms of coupling and cohesion, it is these quantities that must be managed if a system's structure is to be stable and resilient in the face of change, supporting natural growth and evolution, as well as out of the box fitness for purpose and buildability.

On the whole a designer should strive to minimise dependencies between classes in a system, and in particular between packages. This should not be at the cost of making classes uncohesive. They should be as loosely coupled as is meaningful, and this will lead to a more supple component structure. In turn this should lead to a more maintainable and stable system. Where something is stable it can be depended upon without concern.

In an object-oriented system dependencies come in the form of object relationships, so that one class refers to instances of another (dependency, association, aggregation and composition), and class relationships, so that one class inherits from another (or realises an interface). Inheritance is by far and away the stronger dependency, and its use should be subject to careful guidelines.

## Header Coupling

- Unnecessary header inclusion is easy to deal with...
  - ♦ Remove the surplus *#includes*
- But what about uncohesive headers?
  - ♦ Header files often grow by holding unrelated features



```
file.hpp

class stream
{
public:
    virtual ~stream();
    ...
};

class file : public stream
{
public:
    file();
    ...
};
```

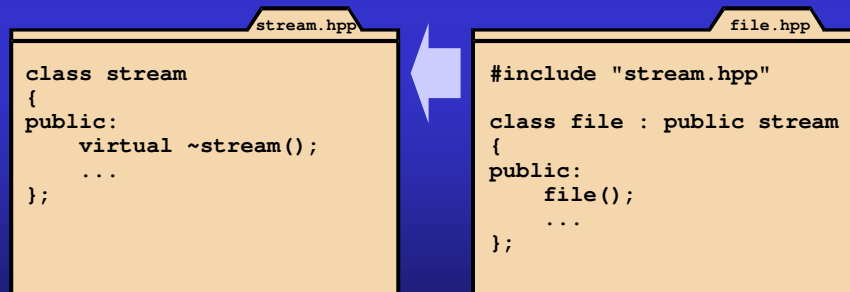
It is possible for a programmer to develop a small project in a single file. However, although this apparently simplifies tool usage — compilation, linkage, versioning, etc. — it does not scale well: comprehensibility falls rapidly with size, editing becomes increasingly awkward, and opportunities for developer teams, parallel development, separate compilation, fine-grained versioning, reuse, unit testing, etc. are all but eliminated. Such a seductively easy route leads quickly to a dead end.

Separate source files in part resolve these issues, but introduce the need for different source files to share common function declarations and type definitions. The temptation is to revert to a monolithic header file that includes all declarations and definitions needed across a project. As with the single source file approach, such a "project.hpp" requires a particular presence of mind to ensure the internal organisation of the file does not degenerate into cryptic spaghetti, a secret dish known only to those initiated in its history and development.

Dividing the declarations and definitions of a project across many header files increases the opportunity for separate development, and many of the benefits that it brings. However, partitioning alone is insufficient to bring in all of the benefits, and may also be used inadvertently as a tool to reduce comprehensibility: The partitioning may be coincidental or inconsistent, making the apparent structure of the system difficult for a programmer to navigate, modify, and extend. The resulting header files may be large and still difficult to comprehend, offering apparently unrelated concepts in a single place.

## Uncohesive Headers

- Split a header file by extracting cohesive features into a new header
  - ♦ Can now reduce unnecessary inclusion



Organise a header around a single concept, such as a class or set of services. The header should declare or include everything it needs to be self-contained, but no more: It should also be a minimal header. There should be a correspondence between source and header file to ensure that the implementation of features offered in such a cohesive header is easy to find.

A cohesive header represents a physical interface that corresponds to the logical structure. It represents a unit of common release and change. For instance, a header may be organised around a single class and its dependent functions. It would include any associated operators and algorithms, helper classes, as well as its own inclusion of any headers whose contents it depended on:

- Define a new empty header with the appropriate name to describe the features to be factored out, including also header guards, and any other common header paraphernalia.
- Move the relevant features from the original header to the new one.
- Have the original header include the new header.
- Compile to ensure that the declarations have all been correctly captured.
- Modify clients, where relevant, to include new header.
- Build to ensure that the dependencies have been captured correctly.
- Move the definitions from the original source to a new source file corresponding to the new header.
- Build and test to ensure that all has gone as expected.

# Forward Declarations

- Problem...
  - ◆ How can a class be referred to in a header without causing its full definition to be included?
- Solution...
  - ◆ Declare a class name rather than include its full definition in a client header file
  - ◆ Include the full header only when needed

```
class stream;  
...  
void copy(stream & from, stream & to);
```

How can a class be referred to in a header without causing its full definition to be included for compilation? Not all types that are included in a header are fully used in that header, e.g., a type name may be used to declare a prototype of as a pointer data member. For a class included for use as a base class or whose instances are manipulated in inline or template functions, the full definition is clearly required. For these other cases, where only a mention of the name is required, inclusion of the full definition seems wasteful.

It is possible to forward declare a `class` or `struct` name for use in function prototypes, `class static` and `extern` data declarations, pointer or reference data members, and some template parameters. However, this does not apply to other type names, e.g., `enum` or `typedef` names. Thus, for classes (and `structs`) declare only the name in the header file, placing it after any included files. In the defining source file include the appropriate header file for all forward-declared types that are used. A header containing only forward declarations simplifies forward declaration for complex type names, such as namespaces with template classes that have defaulted parameters.

Forward declarations support minimal headers based on the principle of sufficiency, leading to improvements in build times and reduction of recompilation when headers are changed. The onus is now placed on the file including the header file to include the full definition, assuming it is needed (not always the case).

## Opaque Types

- The use of forward declarations can be taken a step further...
  - ♦ Type can remain opaque everywhere but the source file in which its associated functions are defined
  - ♦ This is a modular or ADT-based approach

```
namespace file
{
    struct handle;
    ...
}
```

file.hpp

```
namespace file
{
    struct handle
    {
        ...
    };
    ...
}
```

file.cpp

It is possible for a programmer to develop a small project in a single file. However, although this apparently simplifies tool usage — compilation, linkage, versioning, etc. — it does not scale well: comprehensibility falls rapidly with size, editing becomes increasingly awkward and COBOL-like, and opportunities for developer teams, parallel development, separate compilation, fine-grained versioning, reuse, unit testing, etc. are all but eliminated. Such a seductively easy route leads quickly to a dead end.

Separate source files in part resolve these issues, but introduce the need for different source files to share common function declarations and type definitions. The temptation is to revert to a monolithic header file that includes all declarations and definitions needed across a project. As with the single source file approach, such a "project.h" requires a particular presence of mind to ensure the internal organisation of the file does not degenerate into cryptic spaghetti, a secret dish known only to those initiated in its history and development.

Dividing the declarations and definitions of a project across many header files increases the opportunity for separate development, and many of the benefits that it brings. However, partitioning alone is insufficient to bring in all of the benefits, and may also be used inadvertently as a tool to reduce comprehensibility: The partitioning may be coincidental or inconsistent, making the apparent structure of the system difficult for a programmer to navigate, modify, and extend. The resulting header files may be large and still difficult to comprehend, offering apparently unrelated concepts in a single place, i.e., "kitchensink.h" (because they contain "everything but the kitchen sink").

## Representation Coupling

- How can the representation of a concrete class be fully hidden?
  - ♦ Values are fine-grained objects represented through concrete types
  - ♦ Representation may vary

```
class string
{
    ...
private:
    char * text;
    size_t used;
    size_t reserved;
};
```

C++'s model for protection ensures that, from a written source perspective, clients of a class are not — and cannot become — dependent on its internal representation. However, although they cannot access the private section it is still visible in the source code. This can create compilation and binary dependencies: the types of its data members may differ and the `sizeof` an object may change.

In addition, it exposes some of the implementation to the user who may be able to second guess the workings of the class, or gain access by more malicious means (e.g. insert their own `friend` declaration or `#define private public`). As well as being open to such terrorist action, it may be undesirable to distribute the exposed internal workings of a product, e.g., where the same header file is distributed for different releases or platform versions of a product.

For an example of independence of representation, consider that strings are often left unmodified: temporary copies are often created in complex expressions. The plain version of the string will always allocate on construction and deallocate on destruction. Heap activity through creation of duplicate content can be reduced by sharing the representation between copies, releasing it only when there are no outstanding references to it. Reference counting can reduce the cost, but is appropriate only for single-threaded environments. In other words, what may be a link- (or load-) time decision must be taken at compile time because of conditional compilation dependencies.

## The Cheshire Cat Idiom

- Remove everything until there is nothing left but the smile...
  - ◆ Fully hidden representation
  - ◆ Same definition for different implementations
  - ◆ Fewer build dependencies



```
class string
{
    ...
private:
    struct body;
    body * self;
};
```

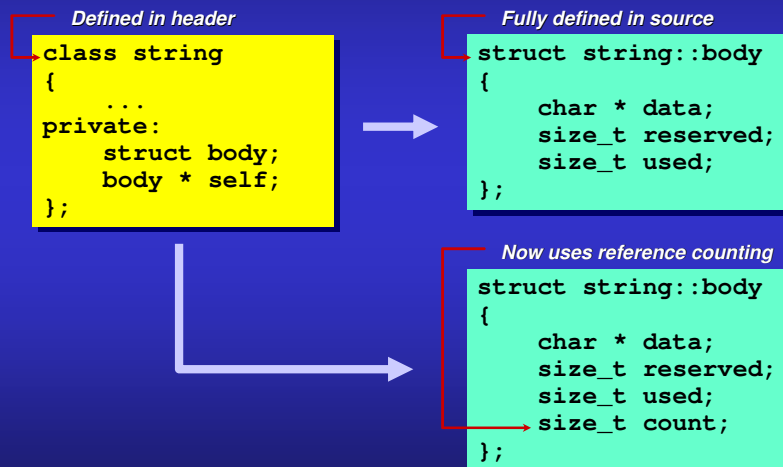
```
struct string::body
{
    char * text;
    size_t used;
    size_t reserved;
};
```

Cheshire Cat [Murray1993] is one of the oldest recognised C++ idioms. It was found originally in the Glockenspiel class libraries. Herb Sutter refers to Cheshire Cat as the Pimpl (*Pointer to implementation*, i.e., `pimpl`) idiom. This naming emphasises the instance representation aspects, i.e., the state of an object, but not other features of a class that are private: Private member data is not the only part of the private section that can benefit from this treatment: private types, private member functions, and private `static` data and functions can similarly be associated with the body rather than the handle class. Thus, Cheshire Cat can use handle–body separation for management of both instance and `static` features.

The solution is to wrap up the real representation within a class (or `struct`), using only a pointer to the forward declared representation type. This removes the body from the class definition, leaving only the 'smile'. The full definition of this private type is provided in the relevant implementation source file where the member function definitions can access it.



# Many Ways to Smile



For an example of independence of representation, consider that strings are often left unmodified: temporary copies are often created in complex expressions. The plain version of the string will always allocate on construction and deallocate on destruction. Heap activity through creation of duplicate content can be reduced by sharing the representation between copies, releasing it only when there are no outstanding references to it.

A counted body resolves this by adding a reference count to the body and the string handles collaborate to increment and decrement it appropriately, ensuring that it is deleted when there are no outstanding references, and forcing a copy on write when they would otherwise modify a shared body. Rob Murray, in *C++ Strategies and Tactics*, illustrates such a migration from an ordinary Cheshire Cat to a reference counted version, illustrating at the same time an immediate benefit of Cheshire Cat in terms of binary compatibility: no recompilation is necessary, only relinking, which may be static or dynamic.

## Inline Function Coupling

- Inline functions are often attractive for their apparent brevity
  - ◆ Easy to write in a header file at the same time, or even in the same place, as the declaration
- However, they increase build times and rebuild frequency
  - ◆ This is often not avoidable for templates, but for non-templated classes there is a remedy
  - ◆ And they are not always an optimisation, because they can increase generated code size

Inline functions are tempting because they are easy to write at the same time and in the same place as the declaration for the function. This is particularly true of accessors — *getters* and *setters* — in a class. These often end up defined in the body of the class definition. The `inline` keyword is often seen by many to mean optimisation. However, this is illusory: It is merely a suggestion to, not a requirement of, the compiler. Inline functions can end up causing code bloat and reducing the runtime performance of a program; quite the opposite of what may be expected. The one thing about them that is certain is that they increase build times and rebuild frequency. Changes to the implementation of an inlined function will cause a rebuild of header dependencies. They may also require additional header files to be visible for their implementation. Inlined functions should be used sparingly. They should be introduced after the need has been demonstrated, not in anticipation of a need. However, there is a great deal of code written that did not have the benefit of this perspective.

# Making Inline Out-of-Line

```
class string
{
public:
    void clear()
    {
        erase(begin(), end());
    }
    ...
};
```

①

*Initially ensure that all inline functions are physically defined outside the class definition...*

*Then take the definitions and move them into the source file, less the inline specifier.*

```
class string
{
public:
    void clear();
    ...
};

inline void string::clear()
{
    erase(begin(), end());
}
```

②

```
class string
{
public:
    void clear();
    ...
};
```

③

```
void string::clear()
{
    erase(begin(), end());
}
```

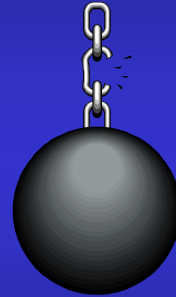
③

Header files plagued by inlined functions can be freed of them by a simple refactoring:

- Move an inline function from within a class definition to later in that file, or into the corresponding header for inlines, if there is one.
- Adjust its signature to include class name and the `inline` keyword.
- Build and test.
- Move the inline function definition into the source file, removing the `inline` keyword.
- Build and test.
- Repeatedly apply this to each function no longer required to be inline, assuming that rebuild costs are not prohibitive, otherwise group many changes together.

# The Fragile Base Class Problem

- Derived classes have a strong dependency on their base classes
  - ♦ Not as fully encapsulated or decoupled as delegation-based relationships
- Base class evolution is a problem...
  - ♦ Release to release binary compatibility
  - ♦ Private changes lead to public builds
  - ♦ Derived class semantics when base is modified



There is a strong structural dependency upward from derived to base classes: a base class cannot be defined without full reference to its base(s). Modifications to a base class can therefore lead to something of a ripple effect especially if implementation is being inherited. This is known as the *fragile base class problem*.

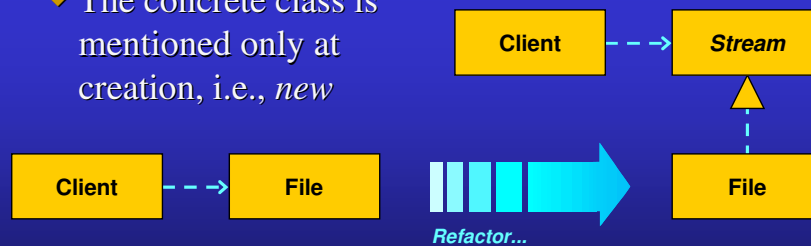
There are two different aspects of the problem: the syntactic fragile base class problem and the semantic fragile base class problem.

The syntactic problem is concerned with both source code and binary compatibility effects of modifying a base class. For instance, the use of `protected` is often questioned, especially for data. This creates even stronger dependencies between derived and base classes. In systems that have wide distribution, such as frameworks, this is almost equivalent to making the data `public`! The use of `protected` should be considered carefully, with ordinary instance data considered the least acceptable feature for this visibility. C++ programmers should note that `private virtual` functions can be overridden and may therefore create a dependency.

What arises from this analysis is an understanding that inheritance and polymorphism should be used principally for defining and developing against interfaces, with an emphasis on establishing stability in the interface's features and its meaning. Reuse of implementation is better left to delegation techniques.

## Interface Decoupling

- Reduce client dependencies on non-public features in a class by...
  - ♦ Separating an interface class from the concrete class, so client code uses pointers to the base
  - ♦ The concrete class is mentioned only at creation, i.e., *new*



It is often the case that developers slap an interface onto an implementation as an afterthought. Such an approach stems from the view that an interface does not actually "do anything", and therefore procedural code is more significant both in terms of its effect and bulk. Hence less effort is invested by developers in writing the interface.

Although an interface requires much less code to express it than an associated implementation, this is perhaps inversely proportional to its relative significance. The interface is the point of agreement between a component supplier and consumer, and should therefore be well considered, complete, comprehensible and stable. Such a state of affairs cannot be reached through casual coding. Bugs in an implementation may be irritating, but they are failures of a component to satisfy an interface, and they can be fixed without adversely impacting clients. Modifications to poorly designed interfaces, however, will break clients that have been written against (and worked around) them; such changes will be seen as causing problems rather than fixing them.

A client can only use the public code of a class, but a class definition typically includes more than simply the public usage interface. This can lead to unnecessary dependencies on users of a package. Interface decoupling introduces a 'vertical' separation between the usage type of an object, i.e., interface, and its actual underlying class. The knowledge of the actual class is required in only one place, i.e. at the point of the creation, leaving all other usage code unaffected.

## Generic Decoupling

- Templates allow logical concepts to be more loosely coupled
  - ♦ However, the use of templates can both increase and decrease physical coupling
- Although a full definition is often in a header, there is no dependency on declared interfaces
  - ♦ Dependency is on usage syntax of a parameter

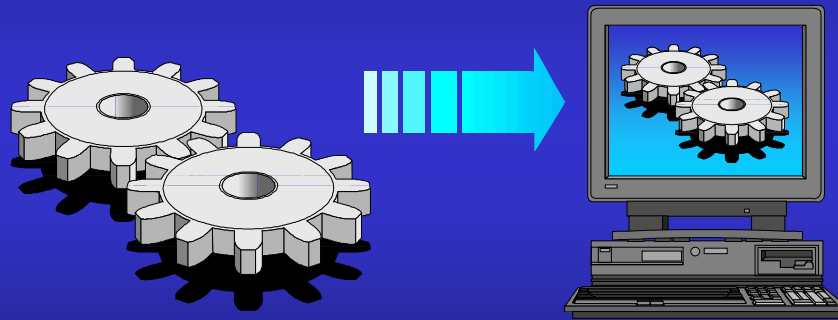


It may seem counterintuitive, but there are scenarios in which templates reduce the coupling in a system. Counterintuitive because templates normally impose the physically coupled burden of including implementation detail in header files. However, as with inheritance (the use of interface classes containing only pure `virtual` functions), the mechanism can be used both to increase and to decrease explicit dependencies.

## Summary

- Dependency management is an architectural concern for any system, especially in C++
  - ◆ Buildability and changeability can be severely hampered if dependencies are unmanaged
- C++ supports many decoupling approaches...
  - ◆ Organisation of header files
  - ◆ Forward declarations
  - ◆ Interface classes
  - ◆ Template parameters

# Policy-Based Design Basics

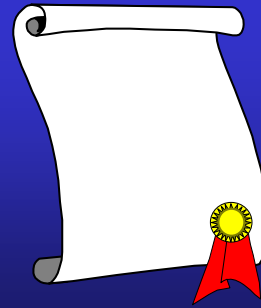


*C++ Advanced*



# Policy-Based Design Basics

- Objectives
  - ◆ Introduce language mechanisms and design concepts that support a policy-based approach
- Contents
  - ◆ Template specialisation
  - ◆ Type traits in the standard library
  - ◆ Default template parameters
  - ◆ Associative container ordering
  - ◆ Sequence adaptors
  - ◆ Allocators



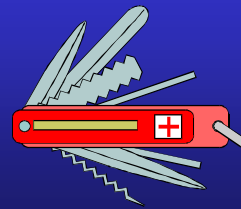
## Parametric Polymorphism

- Parametric polymorphism based on templates supports common generic programming
  - ◆ Expression of containers, algorithms and smart pointers decoupled through an abstracted interface
- It also supports a style of programming based on more complete parameterisation of classes
  - ◆ Policy classes can be used to define interface or implementation
  - ◆ Decisions can be made on types by types

The use of policy parameters to define implementations and interfaces is akin to passing an argument to a function to parameterize its behaviour, based on an interface of virtual functions. The obvious difference is with respect to binding time and scope: policy-based programming is at compile time and often (but not always) at class level; strategy-based programming is at runtime and either at function or at object level.

# Template Specialisation

- Class templates can be fully or partially specialised with respect to their parameters
  - ♦ A specialisation provides an alternative implementation for a particular parameter types
  - ♦ Fully templated version is known as the *primary template* and must appear before specialisations
- Useful for defining traits
  - ♦ Compile-time type-based feature lookup, e.g., `std::numeric_limits`



A class template may be specialised for particular substitutions of its parameters. A full specialisation is effectively a normal class as there is nothing left to parameterise. A partial specialisation lies between the full genericity of the primary template and the specificity of a full specialisation.

## *std::vector<bool>*

- A specialisation for *vector<bool>* has been provided for space optimisation
  - ♦ However, it fails to satisfy container requirements
  - ♦ It is sometimes a pessimisation but cannot be turned off, so consider *deque<bool>* or *bitset*

```
template<typename Allocator>
class vector<bool, Allocator>
{
public:
    ...
    class reference {...}; // smart reference
    reference operator[] (size_type);
    bool operator[] (size_type) const;
    ...
};
```

In standardisation circles this specialisation has acquired the nickname "the evil `vector<bool>`". The intent was that it should provide a packed space-efficient representation. However, it requires a smart reference to act on behalf of `bool`, which means that it does not satisfy container requirements. The reference does itself few favours by also including distinctly non-reference features, such as a `flip` member.

There is no way to turn off this "optimisation", which is unfortunate as it is not always a performance optimisation even if it might be a space optimisation — and priority varies according to need. If this is an issue, use `std::vector<int>` (or some other integral type) or `std::deque<bool>`.

## Looking up Type Traits

- Template specialisation is useful for defining traits related to types
  - ♦ Compile-time type-based feature lookup

```
template<typename NumericType>
struct numeric_limits; // primary template
template<>
struct numeric_limits<int> // full specialisation for int
{
    static int min() throw()
    {
        return INT_MIN;
    }
    ...
    static const bool is_signed = true;
    static const bool is_integer = true;
    ...
};
```

A trait allows compile-time lookup of features related to a type. It is a compile-time form of reflection. Template specialisation supports such type-based selection.

In the example above, the standard defines a `numeric_limits` trait template that is specialised for all built-in numeric types. The syntax for use no longer depends on the type name, e.g., `INT_MAX` versus `UINT_MAX`, and is regular, if a little cumbersome:

```
... std::numeric_limits<type>::max() ...
```

Frequent use of traits for a particular type suggests that a helper `typedef` can be of use:

```
typedef std::numeric_limits<int> int_traits;
... int_traits::max() ...
```

## *std::iterator\_traits*

- Traits are needed for type lookup to declare or implement some functions and classes
- By default, class-based iterators are assumed
  - ♦ I.e., *typename iterator\_type::iterator\_category*

```
template<typename IteratorType>
struct iterator_traits
{
    typedef ... iterator_category;
    typedef ... value_type;
    typedef ... difference_type;
    typedef ... pointer;
    typedef ... reference;
};
```

The following is the definition for the primary `std::iterator_traits` class template:

```
template<typename IteratorType>
struct iterator_traits
{
    typedef typename IteratorType::iterator_category
        iterator_category;
    typedef typename IteratorType::value_type
        value_type;
    typedef typename IteratorType::difference_type
        difference_type;
    typedef typename IteratorType::pointer
        pointer;
    typedef typename IteratorType::reference
        reference;
};
```

## Iterator Tags

- Empty tag types exist that correspond to each iterator type
  - ♦ Used, in conjunction with traits, to select the most appropriate algorithm at compile time

Although, strictly speaking, a forward iterator is also an output iterator, there are no useful applications of a *forward\_iterator\_tag* that derives from an *output\_iterator\_tag*

```
struct input_iterator_tag {};  
struct output_iterator_tag {};  
struct forward_iterator_tag :  
    input_iterator_tag {};  
struct bidirectional_iterator_tag :  
    forward_iterator_tag {};  
struct random_access_iterator_tag :  
    bidirectional_iterator_tag {};
```

Iterator tags are physical manifestations of the iterator categories. Note that although `output_iterator_tag` could be the base for `forward_iterator_tag`, such derivation has few uses. Output iterators are a class apart from the other iterators — write-only concepts are very different to read-only or even read-write concepts.

## *std::iterator\_traits* for Pointers

```
template<typename ValueType>
struct iterator_traits<ValueType*>
{
    typedef random_access_iterator_tag iterator_category;
    typedef ValueType                value_type;
    typedef ptrdiff_t               difference_type;
    typedef ValueType*              * pointer;
    typedef ValueType               & reference;
};
```

```
template<typename ValueType>
struct iterator_traits<const ValueType*>
{
    typedef random_access_iterator_tag iterator_category;
    typedef ValueType                value_type;
    typedef ptrdiff_t               difference_type;
    typedef const ValueType*        * pointer;
    typedef const ValueType         & reference;
};
```

`std::iterator_traits` must be specialised for pointer types as they clearly do not have any `typedef` members. It is because you cannot assume that a type either supports members or is necessarily defined with all the members that you need that traits are such a useful technique.



## *std::iterator* Base Class Template

- Intended for convenience in providing *typedefs*
  - ♦ Declutters class-based iterators slightly

```
template<
    typename IteratorCategory,
    typename ValueType,
    typename DifferenceType = ptrdiff_t,
    typename Pointer        = ValueType *,
    typename Reference      = ValueType &>
struct iterator
{
    typedef IteratorCategory iterator_category;
    typedef ValueType        value_type;
    typedef DifferenceType   difference_type;
    typedef Pointer          pointer;
    typedef Reference        reference;
};
```

To simplify the job of providing the necessary nested types for your own iterator types, the library provides `std::iterator`. Derivation from `std::iterator` represents subtyping, but it does not represent inclusion polymorphism.

For output iterators, all except the first parameter are `void`: it does not make sense to speak of the value type of an output iterator as there is no way you can get your hands on such an object.

## Primitive Iterator-Based Algorithms

- A number of simple algorithms can be considered primitive with respect to iterators
  - ♦ Simple in appearance but useful nonetheless

```
template<typename InputIterator, typename DifferenceType>
void advance(InputIterator &, DifferenceType distance);
```

```
template<typename InputIterator>
typename iterator_traits<Input_iterator>::difference_type
distance(InputIterator begin, InputIterator end);
```

```
template<
    typename FirstForwardIterator,
    typename SecondForwardIterator>
void iter_swap(
    FirstForwardIterator, SecondForwardIterator);
```

The `std::advance` function template advances an iterator by the specified distance. For random-access iterators, pointer arithmetic will be used, but for all others the operation take linear time.

The `std::distance` function template measures the distance from the beginning to the end of the iterator range, i.e. the number of iterations it takes to get from one to the other. For random-access iterators, pointer arithmetic will be used, but for all others the operation take linear time.

The `std::iter_swap` function template swaps the values referred to by its iterator arguments, i.e., `std::iter_swap(lhs, rhs)` is effectively `std::swap(*lhs, *rhs)`.

## Dispatching on Iterator Tags

```
template<typename Iterator> ...  
distance(Iterator begin, Iterator end)  
{  
    typedef iterator_traits<Iterator>::iterator_category type;  
    return distance(begin, end, type());  
}
```

```
template<typename Iterator> ...  
distance(Iterator begin, Iterator end, random_access_iterator_tag)  
{  
    return end - begin;  
}
```

```
template<typename Iterator> ...  
distance(Iterator begin, Iterator end, input_iterator_tag)  
{  
    typename iterator_traits<Iterator>::difference_type count = 0;  
    for(; begin != end; ++begin)  
        ++count;  
    return count;  
}
```

A possible implementation of `std::distance` is shown above. This demonstrates the use of `std::iterator_traits` and the iterator category tag: the appropriate implementation is dispatched to according to the iterator type. Therefore `distance` is constant time for random-access iterators and linear time for all others.

The overloaded `distance` functions are not part of the standard; they are simply possible implementations. You may find that in your library implementation such helpers have been implemented with alternative, less obviously public names.

## Default Template Parameters

- Parameters can be defaulted to simplify class template presentation and usage
  - ♦ Defaults are filled from the right-hand side and cannot be redefined in subsequent declarations
- Useful for providing policy parameters without obtrusively requiring the parameter to be filled
  - ♦ Remember that *typename* must be used to disambiguate types scoped from a template type parameter



The following example:

```
template<
    typename FirstType = void,
    typename SecondType = void>
class example
{
    ...
};
```

Can be used to declare types as follows:

```
example<int, int>
example<int>
example<>
```

Note that the following is not a legal type name:

```
example
```

## Associative Container Policies

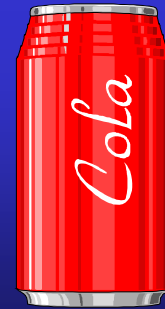
- Standard associative containers have a predicate function object type parameter
  - ♦ Organises elements based on a strict weak ordering
  - ♦ The default ordering is uses *std::less*
  - ♦ Alternative orderings may be parameterised explicitly

```
std::multimap<int, std::string> up;  
std::multimap<int, std::string, std::greater<int> > down;
```

In truth, standard associate containers have two default parameters: the last parameter is the allocator type, which deals with some of the memory management aspects of the container; the penultimate parameter is the comparison type, which deals with the ordering. It is generally safe to assume that the allocator should always be defaulted.

## Adapted Container Policies

- Adapted containers with a simplified interface and policy-driven insertion and extraction
  - ♦ Dispensers can be filled up and can dispense contained items on request
  - ♦ Iteration is not supported
- All the standard container adaptors are dispenser objects that adapt sequences
  - ♦ They default to a suitable sequence for their underlying representation



The standard library provides a number of container types, including three sequence types: `vector`, `deque` and `list`. These offer similar interfaces but different space and time tradeoffs. The library also offers three adapters based on sequences: `queue`, `priority_queue` and `stack`. These offer a protocol for dispensing elements in a particular order, depending on order of insertion and, in the case of `priority_queue`, comparison.

The adapters are not hardwired to a particular underlying container type, just a particular protocol. For instance, `stack` may be parameterised by `vector`, `deque`, `list` or any container type supporting the relevant operations on its last element. Similarly, `queue` may be parameterised by `deque`, `list` or any container type supporting the relevant operations on its first and last elements. The default for each of these adapters is to use `deque`.

## *std::stack*

- A stack is a LIFO (last in, first out) structure
  - ♦ Useful for *undo* lists, parsing and reversing
- Adapts sequences that support *push\_back* and *pop\_back* modifications
  - ♦ Therefore, *vector*, *list* and *deque* are all suitable

```
template<
    typename Type,
    typename Container = deque<Type> >
class stack;
```

An explicit stack type can sometimes be useful to make an ordering policy explicit, but often the underlying types are used because more is needed than the more restricted `std::stack` interface offers.

## *std::stack* Interface

```
template<typename Type, typename Container>
class stack
{
public:
    typedef typename Container::value_type value_type;
    typedef typename Container::size_type size_type;
    typedef Container container_type;
    explicit stack(const Container & = Container());
    bool empty() const;
    size_type size() const;
    value_type & top();
    const value_type & top() const;
    void push(const value_type &);
    void pop();
protected:
    container_type c;
};
```

`std::stack` supports the full set of relational comparisons, subject to their support in the underlying container.



## *std::queue*

- A queue is a FIFO (first in, first out) structure
  - ♦ Useful for working with elements in a given order, e.g., preparing tasks and executing them later
- Adapts sequences that support *push\_back* and *pop\_front* modifications
  - ♦ Therefore, *list* and *deque* are suitable

```
template<
    typename Type,
    typename Container = deque<Type> >
class queue;
```

As with stacks, queues are often expressed as a way that a sequence is used rather than as an explicit type. However, such a type has its uses where only core queue functionality is required and the role of the type is more a matter of communication than of encapsulation.

## *std::queue* Interface

```
template<typename Type, typename Container>
class queue
{
public:
    typedef typename Container::value_type value_type;
    typedef typename Container::size_type size_type;
    typedef container container_type;
    explicit queue(const Container & = Container());
    bool empty() const;
    size_type size() const;
    value_type & front();
    const value_type & front() const;
    value_type & back();
    const value_type & back() const;
    void push(const value_type &);
    void pop();
protected:
    container_type c;
};
```

`std::queue` supports the full set of relational comparisons, subject to their support in the underlying container.

## *std::priority\_queue*

- A priority queue is a structure that ensures its top element is always the largest
  - ♦ Comparison must satisfy strict weak ordering
- Adapts random access sequences that support *push\_back* and *pop\_front* modifications
  - ♦ Therefore, *vector* and *deque* are suitable

```
template<
    typename Type, typename Container = vector<Type>,
    typename Compare =
        less<typename Container::value_type> >
class priority_queue;
```

`std::priority_queue` acts as a sorted container, and the term 'priority' does not indicate anything special. Implementations typically use the `std::make_heap`, `std::push_heap` and `std::pop_heap` algorithm function templates.

## *std::priority\_queue* Interface

```
template<typename Type, typename Container, typename Compare>
class priority_queue
{
public:
    typedef typename Container::value_type value_type;
    typedef typename Container::size_type size_type;
    typedef Container container_type;

    explicit priority_queue(
        const Compare & = Compare(), const Container & = Container());
    template<typename InputIterator>
    priority_queue(
        InputIterator, InputIterator,
        const Compare & = Compare(), const Container & = Container());

    bool empty() const;
    size_type size() const;
    const value_type & top() const;
    void push(const value_type &);
    void pop();
protected:
    container_type c;
    Compare comp;
};
```

`std::priority_queue` does not support any relational comparisons.

## Allocator Usage by Containers

- All container types defined in the standard have an allocator policy parameters
  - ♦ *std::allocator* is the default in all cases
  - ♦ Many container *typedefs* come from the allocator

```
template<
    typename ValueType, ...
    typename Allocator = allocator<value_type> >
class container
{
public:
    typedef Allocator allocator_type;
    allocator_type get_allocator() const;
    ...
};
```

All standard containers are parameterised with respect to an allocator, and they default to the standard allocator. All the constructors for standard containers are overloaded — or use default arguments — to accept allocator objects.

# Allocator Constraints

```
class allocator
{
public:
    typedef std::size_t      size_type;
    typedef std::ptrdiff_t   difference_type;
    typedef value_type       * pointer;
    typedef const value_type * const_pointer;
    typedef value_type       & reference;
    typedef const value_type & const_reference;
    typedef value_type       value_type;
    ...
};
inline bool operator==(
    const allocator &, const allocator &) { return true; }
inline bool operator!=(
    const allocator &, const allocator &) { return false; }
```

*Allocators compatible with standard containers are quite severely constrained in their possible implementation of allocator features.*

Unfortunately, the constraints on allocators used with standard containers are such that there is no point in passing them in. Because all allocators of the same class are required to compare the same and be interchangeable, it means that they are effectively stateless, or at least have only `static` state.

In effect, this makes them quite useless for many purposes to which they might be put: shared memory, persistence, generational pools, etc.

## *std::allocator*

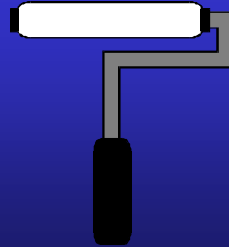
```
template<typename ValueType>
class allocator
{
public:
    ... // constrained allocator requirements, plus...
    allocator() throw();
    allocator(const allocator &) throw();
    ~allocator() throw();
    pointer allocate(
        size_type, allocator<void>::const_pointer hint = 0);
    void deallocate(pointer, size_type);
    size_type max_size() const;
    void construct(pointer, const value_type &);
    void destruct(pointer);
    ...
};
```

*The standard default allocator simply wraps new and delete.*

The standard provides a specialisation for `void`. There is also a `rebind` member that allows, for instance, an allocator specified for a list of `int`, i.e. `std::list< int, std::allocator<int> >`, to actually work with the links used with the list, e.g., `std::list< int, std::allocator<std::list<int>::__link> >`.

## Allocator Issues

- Allocators are hard to implement
  - ♦ And even harder to implement correctly or portably
- Allocators are of relatively little use
  - ♦ No control of memory management strategy
  - ♦ No proper specification of smart memory models
- To address memory management issues, custom containers present a better option
  - ♦ Roll your own



The world of containers would be far simpler without allocators and very, very few people would miss them. People that actually need to customise their memory allocation — for shared memory, for persistence, for the sake of it — find themselves working against rather than with the allocator model.

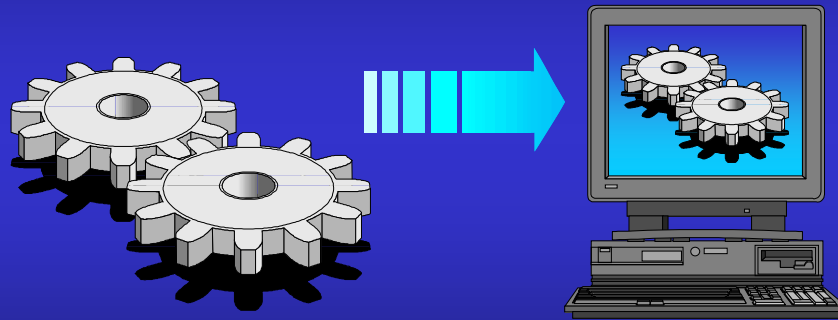
Trying effective memory management of a container whose representation and management is not fully open to you is like eating with a knife and fork... held with chopsticks... through mittens. If you are serious about managing the allocation of a container, then get serious and manage it: write your own container type. It is simpler, more likely to work, and is very much in the extensible spirit of the STL — more so than limiting yourself to the handful of default container implementations in std. It is also more likely to be more portable and more commonly defined. Contrary to popular belief, shared memory allocators subtly depend on how containers are implemented as to whether or not they will work. In other words, even against a portable shared memory API there is no guarantee that code will work between different implementations of the standard library.



## Summary

- Parametric polymorphism offers more than just containers and smart pointers
  - ◆ Template specialisation and defaulted parameters open a wealth of possibilities based on type-level strategies
- Policy-based design is a powerful technique
  - ◆ It is used in the standard library, although not always cleanly and successfully
  - ◆ Adaptation and plug-in behaviour are often the simplest and most successful applications

# Course Outroduction



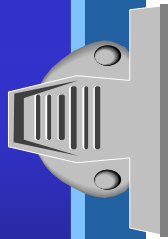
*C++ Advanced*

# Course Outroduction

- Objectives
  - ◆ Wrap up and reflect on the course
- Contents
  - ◆ Check back against the course objectives
  - ◆ Putting knowledge into practice
  - ◆ Moving beyond the course



## Course Objectives



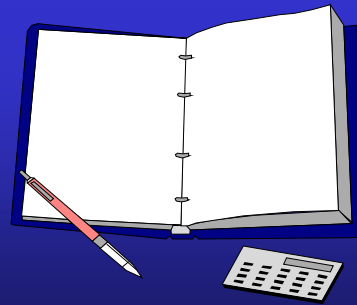
- **Build on C++ knowledge to further introduce the language**
- **Present the key features of the standard library**
- **Practise and extend OO and generic programming skills**
- **Emphasise good practice and safe idioms**



The main objectives of the course have hopefully been met for you, through a combination of lecture material, supporting notes, questions and labs. C++ knowledge has been consolidated and extended with respect to both language and library, as well as in terms of techniques that can be applied that keep code both expressive and safe.

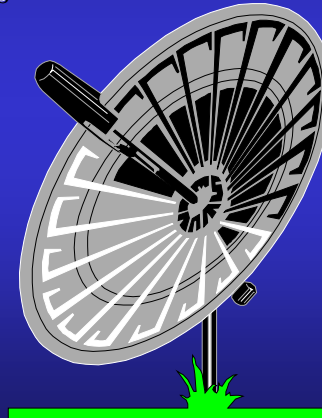
## From Course to Practice

- The course materials provide a resource from which to apply the concepts learnt
  - ♦ Presented slides
  - ♦ Supplied notes
  - ♦ Your own additional notes
  - ♦ Your experience and code from the labs



## Moving Beyond the Course

- The course represents a beginning not an end
  - ♦ To fully appreciate the concepts requires practical experience and further reading
- Follow-on areas include...
  - ♦ Design patterns
  - ♦ Libraries and frameworks
  - ♦ Middleware and components
  - ♦ Development processes



The course includes knowledge useful for understand the more sophisticated aspects of C++ programming. However, to make the most out of this requires practical experience, discussion and further reading, either online or treeware. Even at a more advanced level, to get the most out of C++ involves knowledge of other technologies and practices.

## In Closing...

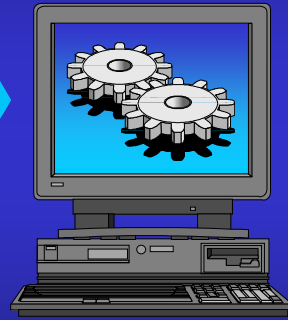
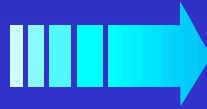
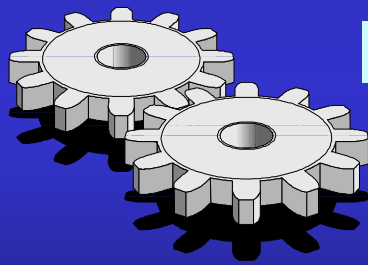
*Any final questions?*



*Thank you and well done!*



C++11



*C++ Advanced*



# C++11

- Objectives
  - ◆ Take a tour of some of the language and library features introduced in the C++11 standard
- Contents
  - ◆ New types and type support
  - ◆ Containers
  - ◆ Memory management
  - ◆ Functional abstraction
  - ◆ Regular expressions and random numbers
  - ◆ Threading

## C++11

- The first major revision of the C++98 standard was approved in 2011
  - ♦ C++03 was a (very) minor update
  - ♦ Its libraries build on TR1
  - ♦ Some new language features displace or simplify usage of many libraries
- C++14 has been approved
  - ♦ This is a minor update in comparison to C++11, and comprises mostly fixes and minor language and library enhancements

## New Fundamental Types

- C++11 has new types and constants
  - ♦ *long long* added for 64-bit integers, also *typedef* support in *<cstdint>*
  - ♦ *char16\_t* and *char32\_t* for UTF-16 and UTF-32 characters, along with *std::u16string* and *std::u32string* typedefs of *std::basic\_string*
  - ♦ *nullptr* is the null pointer constant, of type *std::nullptr\_t*

## Type Support

- `<type_traits>` includes templates for template metaprogramming
  - ♦ Classifiers to determine kinds of types, e.g.,  
`std::is_enum<T>::value`,  
`std::is_reference<T>::value`,
- And additional type support features
  - ♦ E.g., `std::alignment_of<T>::value`, which relates to `alignof` and `alignas` operators

## *auto*

- *auto* specifier has been repurposed to simplify complex declarations
  - ♦ A variable's type can be deduced from its initialiser
  - ♦ A function can be declared with trailing rather than leading return type

```
auto main() -> int
{
    auto answer = 42;
    ...
}
```

## *decltype*

- *decltype* allows the deduction of a type from an expression
  - ♦ Whereas *auto* declares a type based on declaration, and *typeid* is a runtime operation

```
template<typename Lhs, typename Rhs>  
auto divide(Lhs lhs, Rhs rhs) -> decltype(lhs/rhs)  
{  
    return lhs / rhs;  
}
```

## Further Specifiers

- Function and derivation specifiers
  - ♦ *noexcept* and *constexpr* are keywords
  - ♦ *override* and *final* are contextual

```
class base
{
public:
    virtual void polymorphic();
};
```

```
class derived final : public base
{
public:
    void polymorphic() override;
    void non_throwing() noexcept;
};
```

## Aggregate Initialisers

- List initialiser syntax can be used to initialise containers
  - ♦ Based on *std::initializer\_list*
- Aggregate initialisers can also be used for regular object initialisation
  - ♦ But not if constructor is *explicit*

```
std::list<int> c = { 2, 9, 9, 7, 9, 2, 4, 5, 8 };
```



## Range-based *for* loops

- Containers and arrays can be traversed using a simple *for* loop
  - ♦ Syntax is same as Java's *for*-each loop

```
std::vector<int> c = { 2, 9, 9, 7, 9, 2, 4, 5, 8 };  
for(auto i : c)  
    std::cout << i << "\n";
```

```
int c[] = { 2, 9, 9, 7, 9, 2, 4, 5, 8 };  
for(auto i : c)  
    std::cout << i << "\n";
```

## Rvalue References

- An rvalue reference is represented as `&&` as opposed to just `&`
  - ♦ An rvalue will bind to either a *const* `&` or a `&&`, but will favour `&&` if present
- Mainly supports move semantics, which allows copy optimisation

```
std::vector<int> fetch_results();  
std::vector<int> results;  
results = fetch_results();
```

In C++98 (and C++03) the return value of *fetch\_results* is copied and destroyed; in C++11 it is moved, so that there is no effective cost to returning heap-allocating objects by value.

## Unordered Containers

- C++11 provide a family of hashed containers
  - ♦ *unordered\_set*
  - ♦ *unordered\_multiset*
  - ♦ *unordered\_map*
  - ♦ *unordered\_multimap*
- Hashing may be customised through a defaulted policy parameter
  - ♦ Note that the *unordered\_* naming convention avoids clashes with previous implementations that used *hash\_*

# Word Frequency Counter

```
int main()
{
    std::unordered_map<std::string, int> words;
    for(std::string input; cin >> input;)
        ++words[input];
    for(auto frequency : words)
        cout
            << frequency.first << ": "
            << frequency.second << "\n";
    return 0;
}
```

## Fixed-Size Arrays

- The *array* template is for homogenous, fixed-size containers
  - ♦ Offers an STL container interface
  - ♦ No dynamic memory allocation
  - ♦ Relevant to TR1 users, but largely obsolete for C++11 usage
- Supports aggregate initialisation
  - ♦ List initialiser syntax for non-POD types simplifies support in C++11, but still supported for non-C++11

# Hello, Worlds!

```
void say_hello(const std::string & world)
{
    std::cout << "Hello, " << world << "!\n";
}
```

```
int main()
{
    std::array<std::string, 8> worlds =
    {
        "Mercury", "Venus", "Earth", "Mars",
        "Jupiter", "Saturn", "Uranus", "Neptune"
    };
    std::for_each(
        worlds.begin(), worlds.end(), say_hello);
    return 0;
}
```

## Singly Linked Lists

- *std::forward\_list* provides a singly linked list class template
  - ♦ Named *forward\_list* (rather than *slist*) as it supports only forward iterators
  - ♦ *std::list* defines a doubly linked list
- Supports special operations to make up for constraints of single linking
  - ♦ E.g., *insert\_after* rather than *insert*

## *tuple*

- *std::tuple* defines a statically typed, heterogeneous, fixed-size container
  - ♦ Effectively an ad hoc *struct* whose elements are compile-time indexable
  - ♦ Supports lexicographical ordering, depending on its elements
- Can be considered a generalisation of *std::pair*
  - ♦ With which it is closely integrated



## Multiple Return Values

```
std::tuple<int, int, int> today()
{
    std::time_t epoch_time = std::time(0);
    std::tm now = *std::localtime(&epoch_time);
    return
        std::make_tuple(
            now.tm_year + 1900, now.tm_mon + 1, now.tm_mday);
}
```

```
std::tuple<int, int, int> all = today();
int year = std::get<0>(all);
```

```
int year, month, day;
std::tie(year, month, day) = today();
```

```
int day;
std::tie(std::ignore, std::ignore, day) = today();
```

## *unique\_ptr*

- *std::unique\_ptr* can make heap objects behave as scope bound
  - ♦ It is a replacement for *std::auto\_ptr*
  - ♦ Transfers ownership based on move
  - ♦ Can be used in containers
  - ♦ Strictly scope-bound and non-transferring if declared *const*
  - ♦ Supports customised deleter as template parameter

## *shared\_ptr*

- Common tool for managing shared-object lifetime is reference counting
  - ♦ Smart pointers can track the usage count of a heap-allocated object
- *std::shared\_ptr* originated in Boost
  - ♦ Templated and non-intrusive
  - ♦ By default, *delete* called on object when count falls to zero, but a custom deleter can be installed instead

## *weak\_ptr*

- A need for checkable pointers that do not affect object lifetime
  - ♦ So neither *shared\_ptr* nor raw pointers
- *weak\_ptr* fulfills this role
  - ♦ Can be initialised from a *shared\_ptr*
  - ♦ Can be used to observe objects safely without modifying their lifetime
  - ♦ Can be used to break cycles in reference chains

# Keeping Count

```
void strong_ownership_example()
{
    std::shared_ptr<string> ptr(new std::string("Hello!"));
    {
        std::shared_ptr<std::string> qtr = ptr;
        *qtr = "Goodbye!";
        assert(ptr.use_count() == 2);
        assert(!ptr.unique());
    }
    assert(ptr.use_count() == 1);
    assert(ptr.unique());
    assert(*ptr == "Goodbye!");
}
```

```
void weak_ownership_example()
{
    std::shared_ptr<int> strong(new int(2002));
    std::weak_ptr<int> weak = strong;
    assert(strong.use_count() == 1);
    strong.reset();
    assert(weak.expired());
}
```

## Generalised Binder

- *std::bind* supports binding arguments to function calls
  - ♦ Yields a function object that calls the target function, member function or function object, passing explicit and prebound arguments to it
  - ♦ Displaces many of the weaker function facilities in C++98, e.g., *std::bind1st*, *std::bind2nd*, *std::mem\_fun*

## Polymorphic Wrapper

- Function pointers cannot point to objects or member functions
  - ♦ Difficult to write generalised callback code without wrapping, introducing *virtual* functions or templating
- `std::function` plays role of a generalised function pointer
  - ♦ Result type of `bind` is unspecified, so can use `function` or C++11's `auto`

## The Timer On-Off Problem

```
class heating
{
public:
    void turn_on();
    void turn_off();
    ...
};
```

```
class timer
{
public:
    timer(time time_of_day, ??? action);
    void run();
    void cancel();
    ...
};
```



## A Classic C-Style Solution

```
void turn_on(void * context)
{
    static_cast<heating *>(context)->turn_on();
}
void turn_off(void * context)
{
    static_cast<heating *>(context)->turn_off();
}
```

```
class timer
{
public:
    timer(time, void action(void * context), void * context);
    ...
};
```

## A Classic OO-Style Solution

```
class command
{
public:
    virtual void execute() = 0;
    ...
};
```

```
class timer
{
public:
    timer(time, command * action);
    ...
};
```

```
class turn_on : public command
{
    ...
    virtual void execute()
    {
        target->turn_on();
    }
    ...
    heating * target;
};
```

```
class turn_off : public command
{
    ...
    virtual void execute()
    {
        target->turn_off();
    }
    ...
    heating * target;
};
```

## A Simple and Safe Solution

```
class timer
{
public:
    timer(time, std::function<void()> action);
    ...
};
```

```
heating system;
...
timer turning_on(
    when_on, std::bind(&heating::turn_on, &system));
timer turning_off(
    when_off, std::bind(&heating::turn_off, &system));
```

# Lambdas

- Lambda functions are anonymous local functions
  - ♦ They are mapped by the compiler to generated function object types
  - ♦ C++11 language feature displaces experimental Boost Lambda library
- Lambdas can be closures, referring to local variables
  - ♦ Variable capture must be declared

# Lambda Anatomy

*[]* captures nothing  
*[&]* captures all surrounding local variables by reference  
*[=]* captures all surrounding local variables by copy  
*[this]* captures access to surrounding object  
*[&var1, &var2]* captures *var1* and *var2* by reference  
*[var1, var2]* captures *var1* and *var2* by copy

*Optional if empty*

*Optional — deduced from return statements if missing*

```
[capture] (arguments) -> return_type  
{  
    . . .  
}
```

## A Lambda Solution

```
class timer
{
public:
    timer(time, std::function<void()> action);
    ...
};
```

```
heating system;
...
timer turning_on(when_on, [&]{ system.turn_on(); });
timer turning_off(when_off, [&]{ system.turn_off(); });
```

# Hello, Worlds!

```
int main()
{
    std::array<std::string, 8> worlds =
    {
        "Mercury", "Venus", "Earth", "Mars",
        "Jupiter", "Saturn", "Uranus", "Neptune"
    };
    std::for_each(
        worlds.begin(), worlds.end(),
        [](const std::string & world)
        {
            std::cout << "Hello, " << world << "!\n";
        }
    );
    return 0;
}
```

## Regular Expressions

- C++11's a comprehensive regular expression library ultimately comes from Boost
  - ♦ Supports both simple and advanced usage, including subexpression matching and replacement, etc.
- *regex* object 'compiled' from a string
  - ♦ Can be used with search algorithms and iterators, e.g., *regex\_match*, *regex\_search*, *regex\_replace*



# Looking for Something

```
int main(int, char * argv[])
{
    int result = argv[1] ? 1 : 2;
    if(result == 1)
    {
        std::regex expression(argv[1]);
        for(std::string line; std::getline(in, line);)
        {
            if(std::regex_search(line, expression))
            {
                result = 0;
                std::cout << line << "\n";
            }
        }
    }
    return result;
}
```

## Random Number Generation

- The RNG library is based on an orthogonal and extensible design
  - ♦ An engine is a source of (pseudo or real) random numbers expressed
  - ♦ A distribution applies a specific profile to a random number generator
- Note that the C++11 and classic Boost models differ slightly

## Normal Randomness

```
int main(int argc, char * argv[])
{
    int how_many =
        argc > 1 ? std::atoi(argv[1]) : 1000;
    double mean =
        argc > 2 ? std::atof(argv[2]) : 0;
    double sd =
        argc > 3 ? std::atof(argv[3]) : 1;
    std::default_random_engine rng;
    std::normal_distribution<double> bell(mean, sd);
    std::generate_n(
        std::ostream_iterator<double>(std::cout, "\n"),
        how_many,
        [&] { return bell(rng); });
    return 0;
}
```

# Threads

- A thread is an identifiable flow of control in a process, with own stack
  - ♦ Can have own *thread\_local* variables
- A *std::thread* wraps a function or function object
  - ♦ Can provide function plus arguments or a bound function object
  - ♦ Launches on creation
  - ♦ Wait for a thread using its *join* function

## Thread-Related Headers

- Features by header file...
  - ♦ `<thread>` — `std::thread` is like a value type with move semantics
  - ♦ `<mutex>` — mutex and locker types
  - ♦ `<condition_variable>` — condition variables, associated with mutexes
  - ♦ `<future>` — async function result proxy
  - ♦ `<atomic>` — templated `std::atomic` and `std::atomic_flag` types and operations

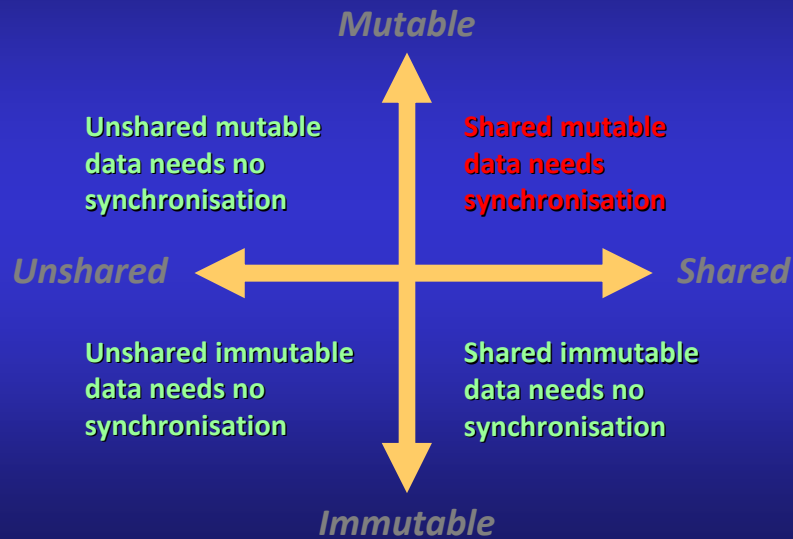
# Mapped Reduction

```
template<typename TaskIterator, typename Reducer>
void map_reduce(
    TaskIterator begin, TaskIterator end,
    Reducer reduce)
{
    std::vector<std::thread> threads;
    for(; begin != end; ++begin)
        threads.push_back(std::thread(*begin));
    for(auto & to_join : threads)
        to_join.join();
    reduce();
}
```

## Thread Safety

- Safety is not a bolt-on quality
  - ♦ Data integrity and program liveness are common victims of incorrectly designed thread interactions
- The unit of thread safety is the function rather than the object
  - ♦ Safety may be achieved by immutability, atomicity or explicit locking

# Synchronisation Quadrant





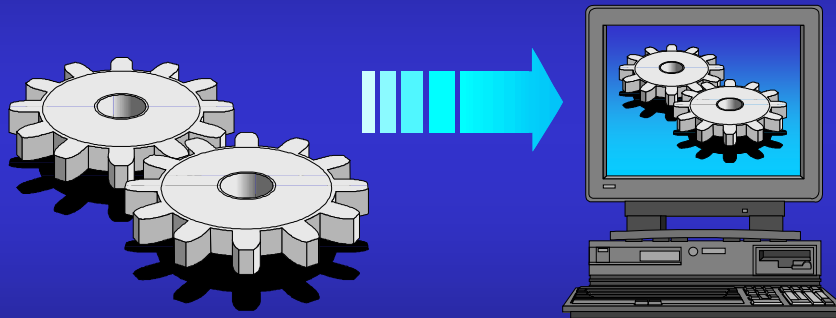
## Critical Regions

- A region of code is critical if concurrent access would be unsafe
- To be safe it must be guarded so no more than a thread at a time runs
  - ♦ A *lock* blocks or lets a thread in
  - ♦ An *unlock* releases next waiting thread
  - ♦ Scope-bound acquisition objects simplify and automate this

## Locking Primitives

- A mutex is a lockable resource for use in defining critical regions
  - ♦ *mutex*, *timed\_mutex*, *recursive\_mutex*, *recursive\_timed\_mutex*, *shared\_mutex*
- A condition variable provide a resource for indicating a condition
  - ♦ A *condition\_variable* indicates a particular state, signalled by *notify\_one* or *notify\_all*

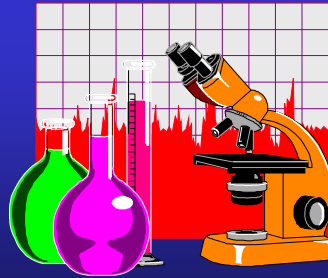
# Labs



*C++ Advanced*

# Labs

- Chapters
  - ♦ C++ Review
  - ♦ Conversions and Mutability
  - ♦ Value-Based Objects
  - ♦ Containers
  - ♦ Encapsulated Algorithms
  - ♦ Exception Safety
  - ♦ Dynamic Resource Management
  - ♦ Dependency Management



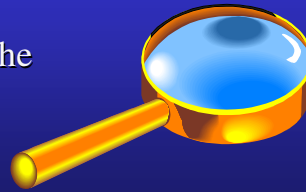
## Lab Guidelines

- Labs follow a number of the chapters
  - ◆ Each lab is made up of a number of different items of different priority:
    - *High priority* (++) : should be completed
    - *Medium priority* (+) : useful to do if you have the time, but not as important as a high-priority requirement
    - *Low priority* : optional and would be nice to do, but definitely not essential
  - ◆ Develop according to requirement priority
    - Aim to complete the high priority items
    - Build up each example incrementally

Consider the high priority requirements to define the bare minimum that you should tackle, and the other requirements to be more optional. Build up each example incrementally, using priority and requirements to define scope of each step.

## Lab: C++ Review

- The principal goal is to develop a couple of classes related to URLs
  - ♦ A simple class to hold URLs
  - ♦ A class to hold a list of unique URLs, as might be used for holding a history of URLs
- An optional goal is to develop a function to write a list of items
  - ♦ Each item is separated from the previous with a comma



The general goals of this opening lab session are to recap on common language and library features that you should hopefully be familiar with — and, if not, this is the ideal opportunity to learn!

# URL

## **++ Requirement 1: Simple URL Usage**

### **Objective**

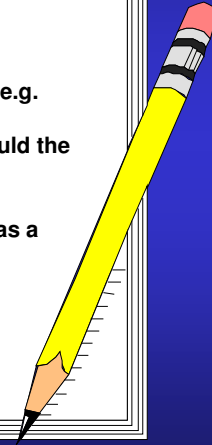
- Provide a class that encapsulates simple URLs.

### **Details**

- The class should support initialisation from a protocol (e.g. "*http*") and a resource (e.g., "*//curbralan.com*").
- The protocol and resource should be queryable, as should the whole resulting URL (e.g., "*http://curbralan.com*").

### **Notes**

- It is reasonable to support default construction as well as a query to determine whether a URL is empty or not.



If you are comfortable with defining `operator==` and `operator!=`, you may also want to consider defining equality and inequality comparison for URLs.

# URL

## ++ Requirement 2: Fixed-Length URL History

### Objective

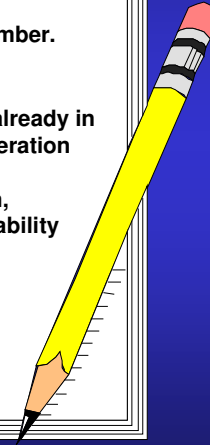
- Provide a class for tracking URLs, up to a maximum number.

### Details

- A maximum history length is provided on construction.
- A URL is only be added to the history list if (1) it is not already in the list and (2) the history list is not already full. The operation should return *true* if a URL is added, otherwise *false*.
- The history list should support queries related to length, whether a URL is already held in the list, as well as the ability to index a URL by position.

### Notes

- Use the *new[]* operator in the constructor to allocate an array of sufficient size for the URLs.
- Ensure that class does not leak resources and is memory safe.



The order of URLs does not matter, so the simplest option is to simply add URLs to the back. Of course, more efficient data structures and algorithms exist to handle the stated requirement, but this exercise is focused on exploring C++ language features rather than optimal algorithm and data structure design.

To ensure that the class is memory safe and does not leak resources, the destructor needs to be defined and some control needs to be taken over the copying behaviour of the class — either disable it or implement a copy constructor and assignment operator allocate and copy correctly, depending on how comfortable you are with these concepts.



# URL

## + Requirement 3: Unbounded URL History

### *Objective*

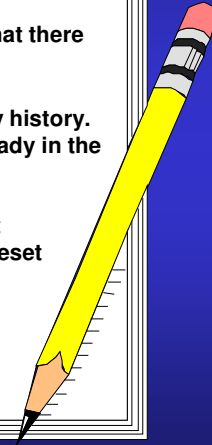
- Modify the URL history class, or define a new one, so that there is no specified upper limit to its length.

### *Details*

- Default construction is supported, resulting in an empty history.
- A URL is only be added to the history list if it is not already in the list. The operation should return *true* if a URL is added, otherwise *false* if the URL was already in the history.
- In addition to common query operations, the history list should support an operation to clear the contents and reset the history to empty.

### *Notes*

- Use a *vector* to hold the URL instances.



The unbounded version should hopefully be simpler to define and make safe, even if you are unfamiliar with the standard `vector`.

# Comma-Separated List

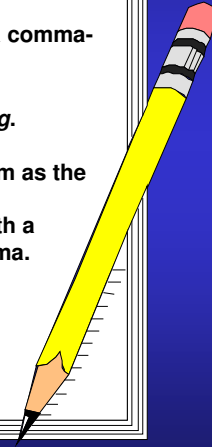
## + Requirement 4: Comma Separation

### **Objective**

- Define a function that turns a sequence of strings into a comma-separated list.

### **Details**

- The function takes a *vector* of *string* and returns a *string*.
- An empty *vector* should result in an empty result.
- A *vector* with a single item should result in only that item as the result.
- A *vector* with many items should separate each item with a comma, but should not terminate the string with a comma.



For example, a vector of the following strings:

```
first  
second  
third
```

Would result in the following string:

```
first,second,third
```

# Comma-Separated List

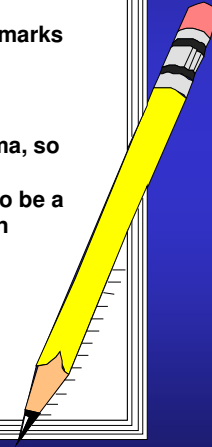
## Requirement 5: Quoting and Escaping

### Objective

- Ensure that strings with embedded commas and quote marks print correctly.

### Details

- If a string contains a comma, in the output it will be indistinguishable from two strings separated by a comma, so use double quotes to surround the whole string.
- A string that contains a double quote mark will appear to be a quoted item in the output, so enclose the whole string in double quotes and double up the embedded quote.



For example, a vector of the following strings:

```
Hello, World!
```

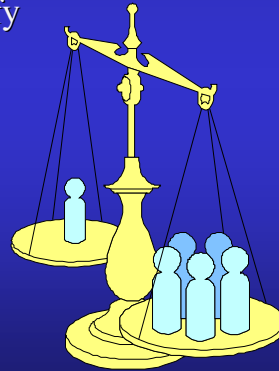
```
He said "Hello"
```

Would result in the following string:

```
"Hello, World!","He said \"Hello\""
```

## Lab: Conversions and Mutability

- The goals of this session are focused on the URL and URL history classes
  - ♦ Supporting conversions to simplify URL usage
  - ♦ Optimise URL history lookup with a cache



This lab session revisits the themes and code of the previous lab session by supporting implicit conversions for the URL class and a caching optimisation for looking up entries in the URL history class.

# URL

## ++ Requirement 1: Conversions to and from URLs

### *Objective*

- Refine the URL class to support convenience conversions to and from strings.

### *Details*

- Support a conversion in the URL class to allow it to be implicitly converted from a string.
- Support a conversion in the URL class to allow it to be implicitly converted to a string.

### *Notes*

- If the source string does not contain a colon, the URL is not well formed and should result in an empty URL object.

In converting from a string into a URL, should conversions from a `const char *` be supported and, if so, what is the simplest way to support this inward conversion? Similarly, in converting from a URL into a string, should conversions to a `const char *` (or even a `char *`) be supported and, if so, what is the simplest way to support this outward conversion?

# URL

## + Requirement 2: Cached URL History Access

### **Objective**

- Optimise the lookup of URLs in the URL history class.

### **Details**

- The last accessed URL is commonly accessed again immediately after either checking its presence or adding it, so internally caching the URL of last access could improve access speed.

### **Notes**

- Because cache access and update occurs in both *const* and *non-const* functions, object state will be changed in *const* member functions.



The current implementation of the URL history class uses a linear search to determine whether a URL is present or not, whether lookup a URL or adding one to the list. Assume that it is common that an accessed URL is likely to be accessed again immediately. Introduce a caching mechanism that is updated on lookup or insertion.

# URL

## Requirement 3: URL Usage Counts

### *Objective*

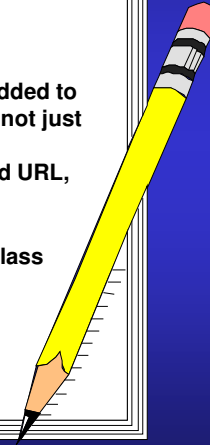
- Keep track of frequency of insertion for URLs.

### *Details*

- Modify the class so that the number of times a URL is added to the history (i.e. how many times it is visited) is tracked, not just whether or not it has been visited.
- As well as caching the most recently looked up or added URL, also track the most frequently accessed URL.

### *Notes*

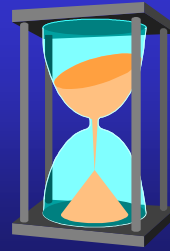
- This requirement both extends the functionality of the class as well as the caching strategy: it should be possible to report usage count and not just presence in the list.



The functionality is being extended, but so to is the caching strategy. The cache now has two parts: the most recently accessed and the most frequently added. Ensure that the cache is use and updated accordingly.

## Lab: Value-Based Objects

- The principal goal is to develop a simple class for date values
  - ♦ Based on the Gregorian calendar, i.e., there is no need to accommodate other calendar models
  - ♦ Supports simple operations, such as querying of year and comparison of two dates
- An optional goal is to develop a simple value class for money
  - ♦ Single currency, supporting arithmetic and comparison



In terms of range, assume that Gregorian dates are astronomical (i.e., independent of locale) and apply to the whole of Common Era (i.e., year 0 onwards). Monetary amounts support unit values and divisions of hundredths, but are otherwise currency agnostic.



# Date

## ++ Requirement 1: Basic Date Usage

### *Objective*

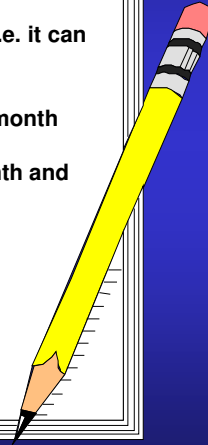
- Define a type for date objects that simply holds a date, i.e. it can be initialised and queried.

### *Details*

- A date object should support initialisation from a year, month and day.
- A date object should support querying for the year, month and the day in the month.

### *Notes*

- Instances of the date type should be copyable.
- There is no need to support default construction.
- There is no need to worry about date validation yet.
- Consider defining a nested *enum* for the month.



Date objects do not really have a reasonable and testable default value, so there is no need to provide a default constructor.

# Date

## ++ Requirement 2: Date Comparison

### Objective

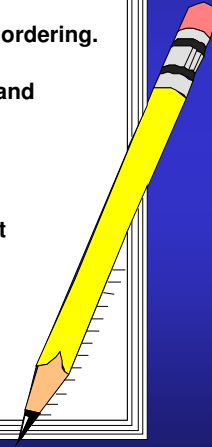
- Allow two date objects to be compared for equality and ordering.

### Details

- Equality comparison is based on providing *operator==* and *operator!=*.
- Ordering comparison is based on providing *operator<*, *operator<=*, *operator>* and *operator>=*.

### Notes

- Consider how to avoid code duplication across different comparison functions.



Equality and inequality operators are likely to contain similar logic. The same is true between the less-than, less-than-or-equal, greater-than and greater-than-or-equal operators. Consider what approaches cut duplication.

# Date

## + Requirement 3: Leap Years

### **Objective**

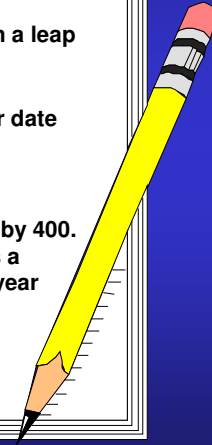
- Allow a date object to be queried as to whether it falls in a leap year or a common year.

### **Details**

- Provide an *is\_in\_leap\_year* function that returns *true* for date objects that represent dates in leap years.

### **Notes**

- A leap year is defined as one that is divisible 4, but is not otherwise divisible by 100 unless it is also divisible by 400.
- For example, 2001 is a typical common year and 1996 is a typical leap year, whereas 1900 is an atypical common year and 2000 an atypical leap year.



If you cannot immediately see what the logic for leap year determination is, a simple and progressive approach to implementing this requirement is to start with the simplest and most common case: write an `is_in_leap_year` implementation that works for typical common years. From there, refine it so that it handles typical leap years, and from there atypical common years, and from there atypical leap years. Finally, see if there is any clean up that can be performed in the code.

# Money

## Requirement 4: Money Usage

### Objective

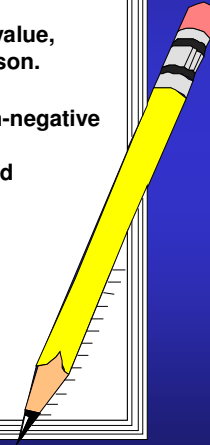
- Define a type for money objects that holds a monetary value, supporting initialisation, query, arithmetic and comparison.

### Details

- A money object should support initialisation from a non-negative unit amount and a non-negative hundredths amount.
- A money object should support querying for the unit and hundredths amount.
- A money object should support equality and ordering comparison with other money objects.
- A money object should support addition, subtraction, multiplication and division.

### Notes

- Money objects should be copyable and default constructible.



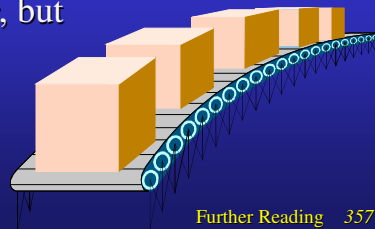
Most currencies support a model of money that divides it into unit quantities and hundredth quantities — pounds and pence, euros and cents, dollars and cents, etc. The money type is not intended to be currency specific, but it is intended for use in a single currency model, i.e., there is no need for each monetary value to be aware of its currency of application.

Some questions to consider in implementing a money type: What is a reasonable default value? What is a reasonable representation that supports simple comparison and arithmetic? In supporting comparison and arithmetic, what families of operators should be supported?

Note that at the moment there is no requirement to worry about validation on construction.

## Lab: Containers

- The principal goal is to develop a collection that holds strings in recently used order
  - ♦ The most recently inserted item is first, the least recently inserted item is last and items are unique
- An optional goal is to develop a sparse container
  - ♦ It appears to be like a *vector*, but if an indexed element does not actually exist, a default value is returned



© Curbralan Ltd

Further Reading 357

Application menus often hold a list of the most recently opened documents and modern phones normally hold a list of the most recently dialled numbers. The principal goal of this lab session is to develop a container class that can be used as a recently used list.

Some collections of values hold a majority of elements that are the same value, e.g. an array of numbers with most values equal to zero. An optional goal of this lab session is to develop a container that appears to hold more values than it actually contains, optimising away values that are equal to some specified default value.

# Recently Used List

## **++ Requirement 1: Core Features**

### ***Objective***

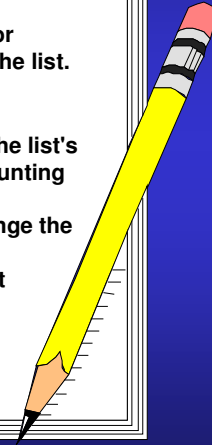
- Provide a public interface and private implementation for inserting strings into a recently used list and querying the list.

### ***Details***

- A recently used list is initially empty.
- Adding an item that is not already in the list increases the list's size by one and places that item in the first position, shunting the others down.
- Inserting an item that is already in the list does not change the size, but it does move the item into the first position.
- It should be possible to index into the list, with the most recently inserted at position 0.

### ***Notes***

- Follow STL conventions for the function interface.



A (most) recently used list (or MRU) should follow STL conventions for its member functions. In many ways it can be considered a hybrid of a sequence (its elements can be accessed by index position) and an associative container (its elements are key-like and unique).

# Recently Used List

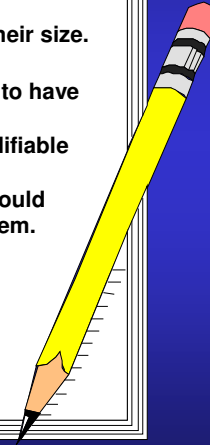
## + Requirement 2: Bounded Capacity

### *Objective*

- Allow recently used lists to have a fixed upper limit to their size.

### *Details*

- On creation, a recently used list may be constrained to have an upper size limit.
- This limit should be queryable, but it should not be modifiable after construction.
- Attempting to add any new items when the list is full should have the effect of dropping the least recently inserted item.



This change will involve adding a new constructor, a new query operation and additional logic in the insertion operation. Note that it is normal behaviour for a list to overflow, it is *not* an error condition.

## Recently Used List

### Requirement 3: Extra Features

#### *Objective*

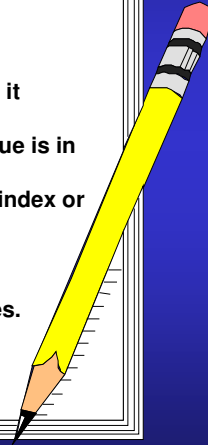
- Make the interface of the recently used list richer.

#### *Details*

- It should be possible to clear a recently used list so that it becomes empty.
- It should be possible to find out whether a particular value is in the list.
- It should be possible to remove an item from the list by index or by value.

#### *Notes*

- You may find that you can refactor some of the existing functionality to take advantage of the new public features.



If there are any other additional features you think the class would benefit from, feel free to include them. However, keep in mind that the style of the class should follow that of other STL containers.



# Sparse Vector

## Requirement 4: Key Features

### Objective

- Provide a public interface and private implementation for inserting items into a sparse vector.

### Details

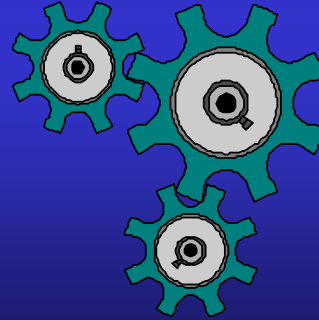
- A sparse vector is templated on the type of the item it holds.
- Entries in a sparse vector are indexed using an integer. For *const* access, if the subscripted item is actually held then it is returned, otherwise a default value is returned.
- On creation, a sparse vector should be provided with a default value to use for entries that have not actually been stored in the sparse vector.
- The size of the sparse vector will appear to grow whenever elements that do not exist are accessed in a non-*const* way.

The public interface of a sparse vector should be a subset of and similar to that of a standard `vector`, and should follow STL container conventions. What container can be used to implement an ordered but discontinuous range of mappings from an integer index to some templated value type?

Note that non-`const` access allows assignment, and this may result in a value being assigned to an element that is the same as the default value. Do not worry about trying to optimise away this case.

## Lab: Encapsulated Algorithms

- The goal is to implement a set-like container with fast lookup and minimal memory usage
  - ♦ Instead of an internal tree representation, a sorted *vector* is used
  - ♦ Inserting new items should not result in resorting the whole container



The standard algorithms in `<algorithm>` can be used to help insertion and lookup. To avoid resorting the whole container on each insertion, consider that if the container is already sorted, the simplest insertion is to place the new element in the right place to start with.

# Flat-Packed Set

## ++ Requirement 1: Core Features

### Objective

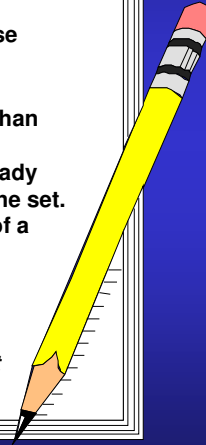
- Provide a class template for a set of unique values whose implementation is based on a sorted *vector*.

### Details

- The template parameter is required to support the less-than operator.
- The *insert* function adds a new element if it was not already included or does nothing if the element was already in the set.
- The *count* function returns the number of occurrences of a value in the set, and therefore zero if it is not in the set.
- Also provide *empty*, *size* and *clear* functions.

### Notes

- Follow STL conventions for the function interface — specifically the interface of *std::set* — but declare *insert* to return *void*.



How can a new element be inserted without resorting the whole sequence? What is the most efficient way to count the number of occurrences of a value in the set?

The default constructor, copy constructor, copy assignment operator and destructor generated by the compiler are all based on the copying of the internal `std::vector`, so there is no need to define any of them at this stage.

# Flat-Packed Set

## + Requirement 2: Custom Ordering

### *Objective*

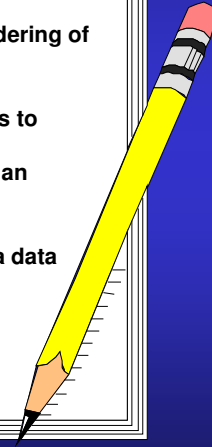
- Accommodate a template parameter that defines the ordering of elements in the flat-packed set.

### *Details*

- The template parameter is a function object that defaults to `std::less`.
- Replace existing explicit and implicit uses of the less-than operator with uses of the function template.

### *Notes*

- Consider holding an instance of the function object as a data member.



Note that there is no need to further generalise and take an allocator parameter.

# Flat-Packed Set

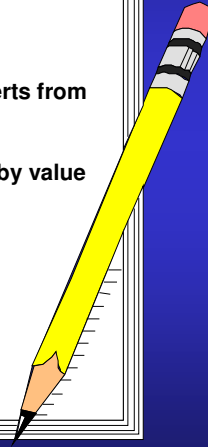
## Requirement 3: Extra Features

### Objective

- Extend the functionality of the flat-packed set.

### Details

- Provide an overloaded *insert* function template that inserts from an iterator range.
- Provide constructors for iterator ranges.
- Provide *erase* functions that allow removal of elements by value or by iterator.
- Provide iteration types and functions.



What other functions does a `std::set` provide that could be provided by your flat-packed set?

## Lab: Exception Safety

- The principal goal is to adapt the date class to support exception handling safely
  - ♦ Invalid date initialisation should cause an exception to be thrown, including from stream input
  - ♦ A date should be refactored to use a handle-body representation that is exception safe
- An optional goal is to also perform validation checking for money



This session revisits the date and money labs from a previous session, extending and refining existing behaviour and implementation.

# Date

## ++ Requirement 1: Date Validation

### Objective

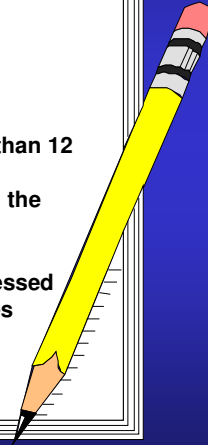
- Ensure that invalid date instances cannot be created or assigned, throwing an exception if necessary.

### Details

- A date is not valid if the year is negative.
- A date is not valid if the month is less than 1 or greater than 12
- A date is not valid if the day in the month is less than 1.
- A date is not valid if the day in the month is greater than the number of days in that month for that year.

### Notes

- Assume that all the kinds of initialisation error are expressed through a single exception type. Multiple exception types are not required.
- A table lookup is the best approach for dealing with the days in the month and day of the year requirements.



This lab assumes that leap year functionality has been implemented. If it has not been implemented, do this first. Once this requirement has been met, it is also a simple matter to include a public query operation for the total number of days in the month represented by the date object.

# Date

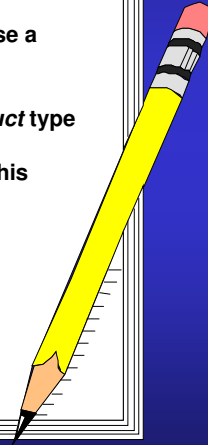
## ++ Requirement 2: Handle–Body Representation

### *Objective*

- Reimplement the internal structure of the date type to use a handle–body representation.

### *Details*

- The existing data members are moved into a nested *struct* type and held indirectly via a pointer.
- Existing functionality is realised by accessing data via this additional level of indirection.
- Constructors, destructors and assignment must handle correctly allocating and deallocating the body object.



Default construction can now be supported easily, so that a null body implies "no date" or "not a date", which is useful for situations where a date is unknown or indeterminate.



# Date

## + Requirement 3: Date I/O

### Objective

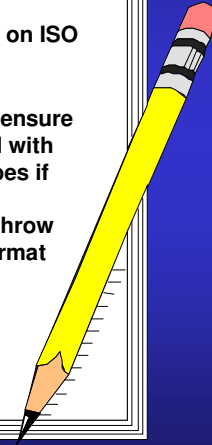
- Support stream input and output for date objects based on ISO 8601 format.

### Details

- A global *operator<<* for *std::ostream* and a date should ensure that the year, month and day in the month are displayed with four, two and two digits, respectively, padding with zeroes if necessary.
- A global *operator>>* for *std::istream* and a date should throw an exception if the date cannot be read in the correct format or if the input does not represent a valid date.

### Notes

- ISO 8601 format is *YYYY-MM-DD*.



Perhaps the simplest technique for validating date on input is to read in three integers, create a new object — letting the constructor do all the validation work — and then assign the valid object to the date object the caller passes in. This ensures that the result argument only holds valid dates.

# Money

## Requirement 4: Money Validation

### *Objective*

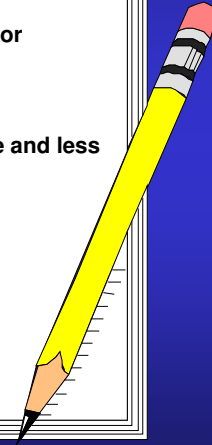
- Ensure that invalid money instances cannot be created or assigned, throwing an exception if necessary.

### *Details*

- The specified unit amount must be non-negative.
- The specified hundredths amount must be non-negative and less than 100.

### *Notes*

- Negative monetary values are still possible because subtraction and negation can be performed on money objects.



Define a nested exception type based on `std::logic_error` for throwing on failed construction.

## Lab: Dynamic Resource Management

- The principal goal is to implement a reference-counted smart pointer
  - ♦ The type is templated and the reference counting scheme should be non-intrusive
  - ♦ An optional disposal action can be provided
- An optional goal is to implement a simple memory tracker for global *new* and *delete*
  - ♦ This can be used for memory leak detection



This lab session involves standalone examples that focus on memory tracking, in one way or another: the main exercise is focused on reference counting, which can be used as a memory-management strategy; the more optional exercise is a simple leak detector.

# Reference-Counted Smart Pointer

## ++ Requirement 1: Non-Intrusive Smart Pointer

### Objective

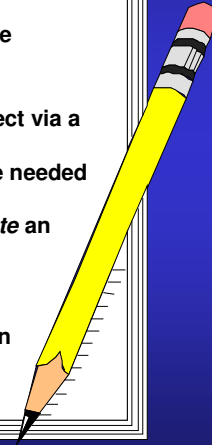
- Implement a templated smart pointer that uses reference counting to share heap-allocated objects safely.

### Details

- A constructor is needed that takes ownership of an object via a pointer.
- A copy constructor and a copy assignment operator are needed for sharing an object.
- A destructor is needed to unshare and, potentially, *delete* an object.
- *operator->* and *operator\** should be provided.

### Notes

- Assignment should be self-assignment safe, i.e. deletion should not occur if assigning from *this*.



The key question concerning the implementation of the smart pointer is which reference-counting technique should be used? The key constraint is that it must be non-intrusive with respect to the templated type.

Further questions to ask yourself, concerning the smart pointer's interface, include: Should default construction be supported? Should the single argument constructor be converting or `explicit`?

# Reference-Counted Smart Pointer

## + Requirement 2: Custom Disposal

### *Objective*

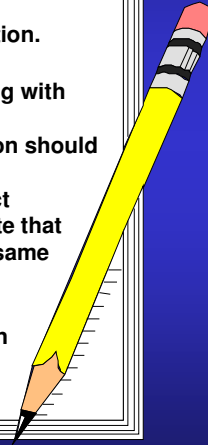
- Support the option of providing an optional disposal action.

### *Details*

- A disposal action can be provided on construction, along with the object to be reference counted.
- When the reference count falls to zero the disposal action should be called on the target object.
- The disposal action can be expressed either as an object derived from a particular class or a function pointer. Note that the argument type of the disposal action should be the same as the smart pointer's template parameter.

### *Notes*

- The default action is still to *delete* the target object when the reference count falls to zero.



The disposal action should only be provided at construction along with the pointer to the object be counted. The disposal action also needs to be passed around with the target pointer. If it is an object, who should own and dispose of the disposal action?

## Instrumented *new* and *delete*

### Requirement 3: Tracking Allocation and Deallocation

#### Objective

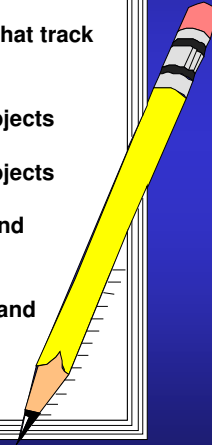
- Provide alternative implementations of *new* and *delete* that track allocation and deallocation.

#### Details

- *operator new* should increment a count of number of objects allocated.
- *operator new* should increment a count of number of objects deallocated.
- Additional functions should be provided to both reset and query the current counts.

#### Notes

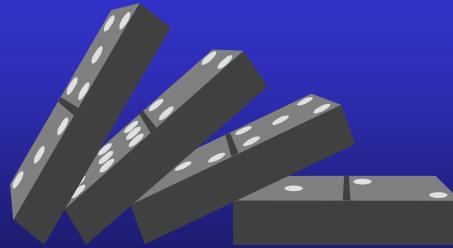
- *operator new* should use *malloc* for allocating memory and *operator delete* should use *free* for deallocation.



The facility described relies on `static` data, and is therefore not suitable for multithreaded environments. Consider also providing additional statistics, such as the minimum, maximum and average size of allocated objects.

## Lab: Dependency Management

- The goal is to decouple the header of the recently used list from the representation
  - ♦ A Cheshire Cat (Pimpl) representation leaves a smaller dependency footprint than a direct data member representation



This lab session revisits the recently used list implementation, with a view to decoupling header file from implementation detail.

## Recently Used List

### **++ Requirement 1: Cheshire Cat List**

#### **Objective**

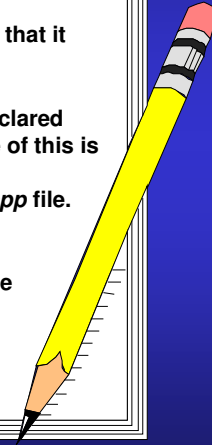
- Refactor the implementation of the recently used list so that it uses a Cheshire Cat representation.

#### **Details**

- The data members are moved into a private, forward-declared *struct*, which is fully elaborated in the *.cpp*. An instance of this is held at a level of indirection.
- All member functions should be "out of line" and in a *.cpp* file.

#### **Notes**

- Make sure that copy construction and assignment are exception safe and that there are no memory leaks in the new implementation.



The new implementation may require that function bodies are moved from a header into a *.cpp* file. The container type should not be visible in the header file and the only header dependency should be on `<string>`.



## Recently Used List

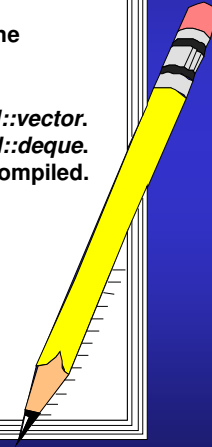
### + Requirement 2: Different Representation

#### *Objective*

- Demonstrate effect of loosened coupling by changing the underlying container used for the list.

#### *Details*

- If `std::deque` is the current container, change to use `std::vector`.
- If `std::vector` is the current container, change to use `std::deque`.
- Rebuild and note which files are (and are not) being recompiled.



Consider the effect of this on a larger project where a common type has had its representation changed.