Advanced C++ Programming

Advanced C++ Programming Contents

Contents **Slide Number**

Advanced C++ Programming Contents

About This Course

- This course overviews some advanced usage of C++ and recent C++ features
  - Visual Studio and GNU History:
    - Visual Studio .NET 2008 – C++98
    - Visual Studio .NET 2010 – C++0x (partial)
    - Visual Studio .NET 2012/2013 – C++11 (partial)
    - Visual Studio .NET 2015 – C++14 (partial)
    - Visual Studio .NET 2017 – C++17 (partial)
    - Visual Studio .NET 2019 – C++20 (partial)
    - Visual Studio .NET 2022 – C++20
    - GNU C++ (depends on version!)

New Features Introduction

- This section covers:
  - Online C++ Resources
  - C++11/C++14 Introduction
  - C++11 and C++14 Features
  - C++11/C++14 Supported Features

Online C++ Resources

- There are many online C++ resources
  - Some allow writing and running code (with a variety of compiler options)

https://www.onlinegdb.com/online_c++_compiler

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

https://wandbox.org/

  - The compiler explorer allows viewing of generated assembly code (compiler options):
    - Especially useful to observer compiler evaluation

https://godbolt.org/

C++11/C++14 Introduction

- C++11 Introduces many language features
  - These can help with type safety and efficiency
  - The Standard Library revised to improve efficiency
    - Many new types and features
- C++ has lacked cross platform APIs
  - Beginning to be address within C++11
    - Threading support
  - Future C++ standards will add additional APIs

C++11 and C++14 Features

- There are many new features in C++11 and C++14 to improve the language and library, e.g.
  - Improve compilation
  - Simplify class definition
  - Simplify template definition
  - Improve program efficiency
  - Improved Standard library

C++11/C++14 Supported Features

- The support for C++11/C++14 varies between different compiler
  - Below are links giving some details:
    - Visual Studio:

http://msdn.microsoft.com/en-us/library/vstudio/hh567368.aspx

http://blogs.msdn.com/b/vcblog/archive
/2014/11/17/c-11-14-17-features-in-vs-2015-preview.aspx

- GCC (add option -std=c++11):

http://gcc.gnu.org/projects/cxx0x.html

- GCC (add option -std=c++14):

http://gcc.gnu.org/projects/cxx1y.html

Using Language Features Correctly

- This section reviews some important features and usage:
  - Const and Casting
  - Casting
    - const_cast
    - static_cast
    - dynamic_cast
    - reinterpret_cast
  - Overloading on Const
  - Logical Const vs Physical Const
  - mutable

Const and Casting

- The use of 'const' within C++ provides for safety and efficiency
  - Safety in that it can help prevent unintended changes to data
  - Efficiency in that it allows compiler optimisation
- Developers should endeavour to provide 'Const Correctness'
  - Make 'const' anything which should not change:
    - Data; Arguments and Declarations

Casting

- C++ now has a number of casting options
  - 'C' style cast **(int) a**
  - Functional style cast **int(a)**
- New style casts:
  - const_cast<T>(…)
  - static_cast<T>(…)
  - dynamic_cast<T>(…)
  - reinterpret_cast<T>(…)

const_cast

- const_cast can be used for casting away constness and volatility.
  - Clearly this should be used with caution, but if necessary!

```
void do_work( const SomeData* sd)
{
   SomeData * temp = const_cast<SomeData*>(sd);

   temp->update( 101);
}
```

Cast away const

Modify Object

static_cast

- Whilst the use of 'static_cast' seems similar to the 'C' style cast it cannot be used as an all purpose cast
  - Does not allow 'const' casting
  - Allows compiler conversions
  - Often used for type promotion:

```
int a = 123;
int b = 71;
double result = static_cast<double>(a)/b;
```

dynamic_cast

- Dynamic cast is for runtime casts and requires RunTime Type Information (RTTI)
  - Typically used for casting within hierarchy

Returns zero if fails for pointers

```
void do_work( ABase *pBase)
{
        ADerived *pDerived = dynamic_cast<ADerived*>(pBase);

        if( pDerived)
        {
                // Work with valid pointer…
        }
}
```

Throws bad_cast exception if failure when casting references

reinterpret_cast

- Cast between different type
  - Not portable as compiler dependent

```
char pch[] = "abcdefgh";

int* pData = reinterpret_cast<int*>(pch);
```


Overloading On 'const'

- There are a number of occasions where overloading on ‘const’ is a common
  - Defining indexers is one of them:

```
class MyArrayWrapper
{
        int data[10];
   public:
        int& operator[](int i) {  return data[i];}

        int operator[](int i) const {  return data[i];}
        …
}
```

Non-const allows modification of data (returns reference)

Const does not allow modification of data (returns value)

Logical vs Physical Constness

- Physical constness implies the memory does not change
  - Values and Objects declared as const!
- Logical constness implies that memory does not appear to change
  - Whilst an object may be declared or passed as a const object, internal implementation allows modification
  - Caching provides a motivation for implementing Logical Const

Logical Const - Cache

- A Cache for resources provides a motivation of Logical const
  - A 'get' used to access an individual resource

Allows call on const object

```
class Cache
{
public:
   shared_ptr<DataSet> get( const string& title) const
   {
            // If dataset with title not already in memory load it

            // return dataset
   }
}
```


mutable

- A previous slide illustrated using a cast to cast away constness
  - This may occasionally may be required
- Where it is necessary to formally allow modification within constant objects it is better to use 'mutable' keyword

```
class Cache
{
   mutable list<shared_ptr<DataSet> > _data;
   …
}
```

Collection could be modified within const methods

Using Language Features Correctly - Summary

- This section gave a review of some important features and usage:
  - Const and Casting
  - Casting
    - const_cast
    - static_cast
    - dynamic_cast
    - reinterpret_cast
  - Overloading on Const
  - Logical Const vs Physical Const
  - mutable

Conversions

- This section gives a refresher on conversions:
  - Conversions Introductions
  - Signed/Unsigned Conversions
  - Expression Evaluation
  - Converting
  - Converting Constructor
  - Explicit Constructor
  - Type Conversion Operator

Conversions Introduction

- C++ has provides conversions for built in types
  - Integral values can be assigned to variables for larger integral type
  - Integral types are value preserving
    - Value preserved, rather than sign
    - Care should be taken assigning to unsigned type

Signed/Unsigned Conversion

- With assignment value is preserved over sign (bit pattern preserved):

```
int a = -1;
unsigned int b = a;

std::cout << b << std::endl;

int c = b;
std::cout << c << std::endl;
```

4294967295

-1

Expression Evaluation

- Evaluation of operators within expressions may involve the conversion of values to 'larger' types
  - Where operands are of different types
    - Operand of 'smaller' type is converted 'larger' type
  - Floating point:

    float > double > long double
  - For integral types, integer promotion

Converting

- The compiler is generally allowed to use one level of user defined type conversion:
  - Converting Constructor or Type Conversion Operator
    - Single parameter constructors can be used for conversion
  - Or
    - Explicit type conversion operator can be define

Converting Constructor

- Constructor can be used for implicit conversion:

```
class SomeData
{
   int _val;
public:
   SomeData(int val):_val(val){}
};
void do_work(SomeData sd){ …}
int main()
{
   SomeData sd = 4;

   do_work(7);
   return 0;
}
```

Implicit call to Constructor

Implicit call to Constructor to create temporary

explicit Constructor

- Constructor can be used for implicit conversion:

```
class SomeData
{
   int _val;
public:
   explicit SomeData(int val):_val(val){}
};
void do_work(SomeData sd){ …}
int main()
{
   SomeData sd(3);

   do_work(SomeData(7));
   return 0;
}
```

Initialisation requires explicit call to Constructor

Explicit call to Constructor to create temporary

Type Conversion Operator

- User define type conversion can be defined:

```
class SomeData
{
   int _val;
public:
   SomeData(int val):_val(val){}
   operator int() const { return _val; }
};
int main()
{
   SomeData sd = 4;

   int result = sd;
   return 0;
}
```

Type Conversion Operator

C++11 allows use of explicit keyword to prevent implicit conversion

Implicit use of Type Converstion Operator

Conversions - Summary

- This section gave a refresher on conversions:
  - Conversions Introductions
  - Signed/Unsigned Conversions
  - Expression Evaluation
  - Converting
  - Converting Constructor
  - Explicit Constructor
  - Type Conversion Operator

Namespaces and Scope

- This section introduces some aspects of namespace and scope:
  - Namespaces
  - Unnamed Namespaces
  - Unnamed Namespaces Example
  - Inline Namespaces
  - Koenig Lookup
  - Static and Extern Variables

Namespaces

- In order to prevent name clashes, names are defined within a namespace:

```
namespace chemicals
{
    class Element
    {...
    };
    class Carbon: public Element
    {...
    };
}
```

Using directive

```
using namespace std;

int main()
{
    chemicals::Element *ps;
    using chemicals::Carbon;

    Carbon crbn(...);
    ...
}
```

Resolving scope

Using declaration

Unnamed Namespaces

- Unnamed Namespaces where consider the superior means of declaring variable or functions with internal linkage
  – Define variables and function within file scope
  – Unnamed namespaces also allow inclusion of types (class and struct)
- Unnamed Namespaces allows easier interpretation of some types of error as names reflect inclusion with ‘unnamed’ namespace

Unnamed Namespace Example

- Example:

```
namespace
{
   int val = 7;
   int square(int a) { return a * a; }

   class Info
   {
      int _a;
   public:
      Info(int a) :_a(a) {}
      int get_a() const { return _a; }
   };
}
```

The use of unnamed namespace allows names to be local to compilation unit

```
int main()
{
   Info data(val);

   int result = square(data.get_a());

   return 0;
}
```

Inline Namespaces (C++11)

- Names defined within an inline namespaces within another namespace will be visible as if within that enclosing namespace:

```
namespace chemicals
{
   class Element
   { //...};
   inline namespace halogens{
      class Chlorine : public Element
      { //...};
   }
}
```

No need to refer to ‘halogen’ namespace

```
int main()
{
   chemicals::Chlorine cl1;

   chemicals::halogens::Chlorine cl2;

   // …
   return 0;
}
```

Koenig Lookup

- Unqualified functions could be defined in many places
  - How is the function found?
- Koenig Lookup or Argument Dependent name Lookup allows finding appropriate functions
  - The types of call arguments are examined
  - Namespaces and classes of arguments are searched for function
    - May be many overloads!

static and extern Variables

```
static int value;


extern int num;



void do_work()
{
   static int count = 0;

   ++count;
}
```

Static variable outside functions have file scope (and initialised to zero) (was deprecated, no longer in C++11)

Extern variable (outside functions) are visible between files

Static variable with functions retain their state between function calls

‘count’ incremented each time function is called

Exception Handling

- This Section covers:
  - Exception Handling Introduction
  - noexcept
  - Exception Functions
  - Exception Safety Guarantees

Exception Handling Introduction

- C++ uses the termination model of exception handling.
- When a problem occurs and an exception is thrown, the flow of execution is terminated.
- The stack is unwound back to the nearest/latest handler for that exception type.

```
void do_work()
{
    try
    {
        throw myexception();
    }
    catch( const myexception& me)
    {
        cout << "Exception no: " << me.what() << endl;
    }
}
```

Any type may be thrown

Only References should be caught (avoids slicing)

Throw Specification (deprecated in C++11)

```
char get_char( const char *cpc, int index)
      throw( out_of_bound)
{
      char ch;

      if( index >= 0 && index < strlen( cpc))
            ch = cpc[index];
      else
            throw out_of_bound();

      return ch;
}
```

Throw specification lists types which function can throw

throw() indicates that a function does not throw any exceptions.

noexcept specifier

- Functions declared with **noexcept** should not throw any exception
  - Results in ‘terminate’ being called if exception thrown!
  - Does not call **unexpected()**
- **noexcept** equivalent to **noexcept(true)**
- Use noexcept instead of throw()
  - throw() is deprecated

noexcept operator

- The noexcept operator is used to allow conditional compilation of template functions
  – Some expansions not throwing exceptions
  – Others allowed to throw exception
- noexcept is **false** if
  – Function called without ‘noexcept’
  – throw expression
  – dynamic_cast,where conversion requires run-time check
  – typeid for polymorphic class

noexcept Example

- Template functions can be defined to conditionally allow exceptions to be thrown!

Arbitrary class with worker function

```
class Test
{
public:
        void worker() noexcept {}
};

template<typename T>
void do_work(T& t) noexcept( noexcept(t.worker())) {
        t.worker();
}
```

Instantiated template function conditionally allows exceptions to be thrown

Exception Functions (C++11)

- New functions have been added to make exception handling more flexible:

Function	**Description**
make_exception_ptr	Wraps an exception object in an exception_ptr
current_exception	Returns an exception_ptr for the current exception object
rethrow_exception	Throws exception for exception_ptr
throw_with_exception	Throws exception, but also nests existing exception
rethrow_if_nested	Throws nested exception

- Some existing functions have been **deprecated**:
  – get_unexpected, set_unexpected

Rethrowing Exception

```
void do_access()
{
   std::vector<int> data{1,2,4,8};
   int result = 0;
   try
   {
       result = data.at(7);
   }
   catch( const std::out_of_range& oor)
   {
       std::exception_ptr ep =
                    std::make_exception_ptr(oor);

       std::rethrow_exception(ep);
   }
}
```

out_of_range exception thrown

Create exception pointer

Rethrow exception

Catch All

```
void do_something()
{
   std::exception_ptr ep;
   try
   {
      get_char( "Hello World", 23);
   }
   catch(...)
   {
      cout << "Caught something!!" << endl;
      ep = std::current_exception();
      std::rethrow_exception(ep);
   }
}
```

Use ... to catch exceptions of any type.
Typically last in list of catch clauses

current_exception captures exception and returns an exception_ptr

Exception Safety Guarantees!

- Exception safety can be set down in a number of ways:
  - No exception safety
  - Basic guarantee:
    - no leakage of resources
  - Strong guarantee:
    - program state always well defined (commit or roll-back)
  - No-throw guarantee

Memory Management

- This section covers aspect of memory management:
  - New Handler
  - Placement New
  - Overloading new and delete

New Handler

- The new handler determines what happens when 'new' fails to allocate memory
- The default implementation will throw a 'bad_alloc' exception
- Can define a custom implementation to perform custom actions:
  - Attempt to free some memory!
  - Defragment memory

set_new_handler

- When the developer takes tighter control over memory allocations and therefore know how memory could be sensibly released
- Old 'new handler' should be retained if intended to reset to original action

```
void my_newhandler()
{
   std::cout << "Problem with memory!" << std::endl;
   throw std::bad_alloc();
}

typedef void(*nhf)();
```

Take appropriate action to attempt to free memory

Typedef for new handler

Using set_new_handler

- Put function in place and keep old function:

```
int main()
{
   nhf oldnewhandler = std::set_new_handler(my_newhandler);
   try
   {
      int* pint = new int[1000000000];

   }
   catch (const std::exception& ex)
   {
      std::cout << ex.what();
   }
   std::set_new_handler( oldnewhandler);
}
```

Returns pointer to old function

Attempted memory allocation

Put original function back in place

Placement New

- Typically when the 'new' operator is used no indication of location within the heap is given for the allocation to take place:

```
int *pData = new int[10];
```

- Placement 'new' allows specifying the location for initialisation of an object within allocated memory:

```
SomeData* psd1 = new (address) SomeData(23,102);
```

Address for initialisation of object

Placement New Example

- Placement new allows custom allocation of objects:

```
class SomeData
{
   int _a, _b;
public:
   SomeData(int a, int b):_a(a), _b(b){}
}
```

```
{
   char *data = new char[10000];
   size_t obj_size = sizeof(SomeData);
   int cnt = 0;

   SomeData* psd1 = new (data + obj_size * (cnt++)) SomeData(23,102);
   SomeData* psd2 = new (data + obj_size * (cnt++)) SomeData(65,77);

   delete[] data;
}
```

Allocating object in sequence!

Calculate position for allocation

Overloading new and delete

- The global operators 'new' and 'delete' are used for dynamically allocating memory and freeing it respectively
- As has already been seen with the placement versions these are overloaded to support custom 'allocation'
- These operators can be completely replaced
  - This should be done with caution as it will replace allocations for all types

New and Delete Signatures

- Global new and delete operator signatures
  - Standard allocators:

```
void* operator new ( std::size_t count);
void* operator new[] ( std::size_t count);
```

  - Standard free:

```
void operator delete (void* ptr);
void operator delete[] (void* ptr);
```

  - Placement allocators:

```
void* operator new ( std::size_t count, void* ptr);
void* operator new[] ( std::size_t count, void* ptr);
```

  - Placement free:

```
void operator delete (void* ptr, void* ptr);
void operator delete[] (void* ptr, void* ptr);
```

Class Specific Allocators

- An alternative (safer option) is to provide class specific new and delete
  - These can be defined in terms of the global allocator and free
  - Signatures of member operators:

```
void* X::operator new ( std::size_t count);
void* X::operator new[] ( std::size_t count);
```

```
void X::operator delete (void* ptr);
void X::operator delete[] (void* ptr);
```

Class Allocators Example

- Class containing definitions of new and delete:

```
class SomeData
{
   int _a, _b;
public:
   SomeData(int a, int b):_a(a), _b(b){}
   void *operator new(size_t size)
   {
      return ::operator new(size);
   }
   void operator delete(void *ptr)
   {
      ::operator delete(ptr);
   }
};
```

Use global new to perform allocation, but could use placement new

Use global delete to perform allocation, but could use placement delete

Templates

- This section covers:
  - Template Functions
  - Special Case
  - Template Class
  - Specialisation Classes
  - Paritial Specialisation
  - Metaprogramming
  - SFINAE
  - C++11 Template Features
  - Koenig Lookup

Template Functions

- Simple Template Function:

```
template<typename T> inline T my_max( T x, T y)
{
   return x < y ? y : x;
}
int main()
{
   double d = 32.3, e = 54.6, f;
   f = my_max( d, e);
}
```

Template function instantiated for function with signature double `my_max`(double, double)

- Parameters type deduction involves 'decay':
  - T will have any const or volatile removed
  - T will be non-reference

Template Parameter Deduction

- Template Parameters are deduced from the argument passed:

```
template<typename T> inline T my_max( T x, T y)
{
   return x < y ? y : x;
}
int main()
{
   double d = 32.3, e = 54.6, f;
   f = my_max( d, e);
}
```

double my_max(double, double)

- The above deduced parameters may not be what is intended, as this implies passing by value

Define As

```
template<typename T>
inline T my_max( const T& x, const T& y)
{     return x < y ? y : x;}
```

Special Case

- Where a special case is required an ordinary function can be supplied. The compiler looks for a match for these functions before instantiating a template:

When specialising Template function

```
template<>
inline const char* my_max(
      const char* x, const char* y)
{
   return std::string(x) < std::string(y) ? y : x;
}
int main()
{
   const char *cpc1 = "Hello", *cpc2 = "there";
   const char *pc;

   pc = my_max( cpc1, cpc2);
   return 0;
}
```

Passing Arrays

- Passing arrays by simple template parameter results in pointer type being used
- Where an array is explicitly expected to be passed as a parameter use the form:

```
template<typename T, unsigned int N>
T total_data( T(&data)[N])
{
   T sum{};
   for (size_t i = 0; i < N; i++) { sum += data[i]; }
   return sum;
}
```

Specialising Classes

- Specialised template class for specific template parameter:

```
template <typename T>
class DataClass
{
};
```

```
template <>
class DataClass<double>
{
…
};

void DataClass<double>::do_work() {…}
```

Specialising on double

Specialised Member Function

Partial Specialisation

- For templates with multiple template parameters is may not be necessary to specialise all type parameters:

```
template <typename T, typename U>
class DataClass
{
  T _data1;
  U _data2;
};
```

Two Type Parameters

```
template <typename T>
class DataClass<T, int>
{
  T _data1;
  int _data2;
};
```

Specialise Second Type as double

Non-type Parameter

- Constants may be used as template parameters
  - These may also have default values:

Default value

```
template<typename T, int size = 10> class DataClass
{
  T _data[size];
public:
  T get_data(int ind) const { return _data[ind];}
  void set_data(int ind, T data) { _data[ind] = data;}
  DataClass(){}
  ~DataClass(void){}
};
```

Use of value within the template class

```
DataClass<double,20> dc1;
DataClass<double> dc2;
```

Metaprogramming

- The use of templates in C++ allows metaprogramming:
  – Values (const) can be evaluated at compile time
  – Recursion can be used with templates

```
template<int val>struct Factorial
{
   enum { Value = val * Factorial<val-1>::Value};
};
template<> struct Factorial<0>
{
   enum { Value = 1};
};
```

SFINAE

- When templates are being instantiated there are typically many options for failure
  - If these resulted in compilation errors, there would be considerable error reporting
    - "Substitution Failure Is Not An Error" (SFINAE)
- SFINAE can also be used to implement Template Metaprogramming

Curiously Recurring Template Pattern

- Abstract diagnostics into separate class:

```
template<typename T> class ACounter
{
   static int _count;
   const int _index;
protected:
   ACounter() :_index(++_count) {}
   virtual ~ACounter() {}
public:
   int get_index() const { return _index; }
};
```

```
class AClass:public ACounter<AClass>{};

template<typename T> int ACounter<T>::_count = 0;
```

Curiously Recurring Template Pattern

Initialisation of static

Static Polymorphism (Template)

- The Curiously Recurring Template Pattern can be used to implement a form of Static Polymorphism:

```
template<typename Actual>
class TheBase
{
public:
  int do_work(int a)
  {
    return static_cast<Actual*>(this)
            ->worker(a);
  }
};
```

```
class Multiple :
          public TheBase<Multiple>
{
  int _val{};
public:
  Multiple(int val) :_val(val) {}
  int worker(int a)
  {
    return _val*a;
  }
};
```

C++11 Template Features

- Many new features enhance the development of templates within C++:
  - Template 'alias'
  - SFINAE – enable_if
  - extern Template
  - Variadic Templates
  - Default Template Function Arguments

Template 'alias'

- Template instantiation can involve the use on long names
  - 'typedef' can be used to create a alternative 'name' for a type
  - C++11 now introduces aliasing as an alternative (which can also be used with templates):

```
template <typename T, int n>
using DC = DataClass<T, n>;
```

At Global Level

  - Use:

```
DC<int, 5> dc(val);
```

SFINAE – enable_if (C++11)

- Conditionally call function dependent upon type trait:

```
template<typename T>
typename std::enable_if<!std::is_class<T>::value, T>::type calc(T a)
{ return a*a; }

template<typename T>
typename std::enable_if<std::is_class<T>::value, T>::type calc(T& data)
{ return data; }
```

```
{
  int a = 100;
  auto result = calc(a);
  std::cout << result << std::endl;
}
```

```
{
  A a;
  auto result = calc(a);
  std::cout << result << std::endl;
}
```

A is a class

extern Templates

- Templates must be defined before they are used
  - Compiler expands template code appropriately for usage – within each compilation unit
  - Compiler may create the same expansion within multiple compilation units – unnecessary work
- Defining template as 'extern' tells the compiler that the expansion is within another compilation unit, thus reducing the amount of work required:

```
extern template class DataClass<int, 5>;
```

Variadic Templates

- 'C' and 'C++' support functions with variable numbers of arguments
- C++11 supports templates with variable numbers of arguments:

```
template<typename T>
T sum(T t) { return t; }

template<typename T, typename... REST>
T sum( T t, REST... rest)
{
        if( sizeof...(rest))
        {
                t += sum(rest...);
        }
        return t;
}
```

Variable number of Types

Variable number of Arguments

Recursive call to Template Function

Variadic Templates Continued…

- The previous example used recursion
- Template functions can be define to evaluate any number of functions on parameters:

```
struct StructEval {
template<typename... Many>
    StructEval(Many...) {}
};
```

Template function to be executed

```
template<typename T>
void funOutput(T t) { std::cout << t << std::endl; }
```

```
template<typename... TheLot>
void evaluate(TheLot... theLot)
{
  structEval( (funOutput(theLot),0)... );
}
```

Parameter Pack Expansion

Default Template Function Arguments

- Whilst classes and thereby member functions could have default template arguments, template free functions could not
- C++11 now allows the use of default template arguments for functions:

```
template<typename R = double, typename T = int>
void worker( R r = R{}, T t = T{})
{
   cout << "First: " << r << endl;
   cout << "Second: " << t << endl;
}
```

Template Template Parameters Motivation

- Template classes may have multiple type parameters:

```
template<typename T>
class Info
{
    T _val;
public:
   Info(T val) :_val(val) {}
   T get_val()const { return _val; }
};
```

```
template<typename T, typename B>
class AClass
{
   B _b;
public:
   AClass(T val) :_b(val) {}
   T get_val() const { return _b.get_val(); }
};
```

```
AClass<int, Info<int>> ac(42);

int result = ac.get_val();
```

Presumes correct use of template parameters

Template Template Parameters

- Template classes may have multiple type parameters:

Template Template Parameter

```
template<typename T,
template <typename> class B>
class AClass
{
   B<T> _b;
public:
   AClass(T val) :_b(val) {}
   T get_val() const { return _b.get_val(); }
}
```

```
template<typename T>
class Info
{
    T _val;
public:
   Info(T val) :_val(val) {}
   T get_val()const { return _val; }
};
```

```
AClass<int, Info> ac(42);

int result = ac.get_val();
```

Template parameter for Info is now first template parameter

Policy Based Design

- This section introduces Policy Based Design:
  - Introduction to Policy Based Design
  - Policy Based Design
  - Example 1
  - Example 1 - Using the Policies
  - Example 2
  - Example 2 - Using the Policies

Introduction to Policy Based Design

- Policy Based Design provides a means of defining types with a wide range of options
  - Potentially many permutations of these options
- Policy Based Design typically relies on the use of Templates
  - Allows options to be compiled in without direct hard coding
- Uses template type parameter as means of providing options
  - Can use inheritance from template type parameter
  - Alternatively parameter type used to define data member

Policy Based Design

- Especially libraries need to provide a wide range of functionality
- Users can be given a wide range of choice over functionality
- A way of allowing user choice is through Policy Based Design
  - Plug in functionality required (Policy)
- Can be implemented using CRT and static polymorphism

Example 1

- Simple example:

```
class APolicy
{protected:
   void the_policy() { std::cout << "A policy..." << std::endl;}
};
class BPolicy: APolicy
{protected:
   void the_policy() { std::cout << "B policy..." << std::endl;}
};
```

```
template<typename policy = APolicy> class Worker: public policy
{
public:
   void do_policy() { the_policy(); }
};
```

Example 1 - Using the Policies

- Using both 'APolicy' or 'BPolicy';

```
{
  Worker<> wap;

  wap.do_policy();

  Worker<BPolicy> wbp;

  wbp.do_policy();
}
```

Default APolicy

Example 2

- Synchronisation example:

```
class NoLockSyncPolicy
{public:
   class Guard
   {public:
      Guard(){ // Do Nothing
      }
   }
};
```

```
class LockSyncPolicy
{public:
   class Guard
   {public:
      Guard() {
         std::cout << "Guarded..." << std::endl;
```

```
template<typename SyncPolicy> class Worker
{public:
   void do_work(){
      auto lock = SyncPolicy::Guard();
      std::cout << "Do Work..." << std::endl;
   }
};
```

Example 2 - Using the Policies

- Using both 'NoLockSyncPolicy' or 'LockSyncPolicy';

Without Synchronisation

With Synchronisation

```
{
  std::cout << "No Lock: " << std::endl;
  Worker<NoLockSyncPolicy> wnl;

  wnl.do_work();

  std::cout << "Lock: " << std::endl;
  Worker<LockSyncPolicy> wl;
  wl.do_work();
  return 0;
}
```

Idioms and Design Patterns

- Value Types
- Operators
- Handle/Body Idiom
- Bridge
- Singleton

Value Types

- This section considers the classification of types
  - Classification of Type
  - Classification
  - Defining Value Type
  - Creating and Destroying
  - Rule of Three

Classification of Type

- Some languages make a clear distinction between some 'type' of type
  - C++ requires that the developer implement the type appropriately for its usage
  - The are difference is the way a type is defined dependent on its intended classification
  - Three distinct classification are:
    - Value, Service or Entity

Classification

- The table below indicates the usage and some aspects of implementation:

Classification	**Purpose**	**Examples**	**Implemented Operations**
Value	Represent simple data, may be wrapper	Number, Point, Size, String	Copy, Compare, various operators
Service	Provide interface to some functionality	CheckStatus	None if stateless
Entity	Identity important and may map to row in database (with primary key)	Person, Employee, Order	Typically none

Defining Value Type

- Implementation
  - Simple data or Wrapper for data (no inheritance)
  - Typically identity is not important
  - Often uses overloaded operators
  - Frequently used directly as parameters and data members
  - Allows copying and assignment
    - Implies passing by value or constant reference

Creating and Destroying

- Value types typically have a default constructor
  - This is a requirement for use in arrays or containers
- Constructors initialising data will also typically be defined
- A destructor will be required if raw pointers are used internally
  - However ideally avoid use of raw pointers

Rule of Three

- The Rule of Three dictates that if one of the three operations **copy constructor**; **destructor** or **assignment operator** is require the all three should be provides (or some prevented)

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

The use of a raw pointer as member would require implementation of Destructor and Copy operations

- With ‘move’, now Five to consider!

Operators

- This Section covers:
  - Operators
  - Assignment operators
  - Defining operators
  - Many special cases
  - Binary operators
  - Type Conversion Operators
  - User Defined Literal

Operators

- Most operators within C++ can be defined for user defined types
- operators provided notational convenience
- operators cannot be redefined for built-in types (e.g. int, char, double)
- programmer **cannot** change:
  - operator precedence
  - order of association, or define
  - :: ?: . .*

Assignment operators

- operator keyword used to define operator
- e.g.

```
class X
{
public:
    X& operator=( const X& x)
    {
            if( this != &x){ …}
            return *this;
    }
};
```

Can return reference when returned object has persistence

Comparing addresses of objects

- usage:

```
X  x1, x2;

x1 = x2; // is equivalent to  x1.operator=(x2);
```

Assignment operators (Preferred)

- operator keyword used to define operator
- e.g.

```
class X
{
public:
      X& operator=( const X& x)
      {
            X temp(x);
            swap( temp);
            return *this;
      }
};
```

Swap member function swaps the members

- usage:

```
X  x1, x2;

x1 = x2; // is equivalent to  x1.operator=(x2);
```

std::swap

- The implementation of assignment illustrates the use of a swap function to swap the bodies
- In order to help in this implementation the std::swap function can be used
- Where members may be 'moved' the std implementation will move the object

Assignment and Copying

- Assignment and Copying should be considered together
  - If one is defined it makes sense to define the other
  - If it is not meaningful to copy then,
    - Common paradigm to not define and make private:

```
class X
{
private:
     X( const X&);
     X& operator=( const X& x);
public:
// ...
};
```

No longer possible to use copy constructor or assignment operator

Defining operators

- Most operators are either unary or binary; there is one ternary operator ?:
- The assignment operator illustrated a binary operator. Here the left hand operand is used as the implicit argument.
- Unary operator member function (the one operand is taken as the implicit argument):

```
class X
{
      X operator++() { X x;… return x;}
};
```

Operators - many special cases

- ++/--
  - operator++() - prefix increment operator
  - operator++(int) - postfix increment operator

- Compound assignment operators
  - **No** special association between = and + to produce +=
  - = and += need to be defined separately
    - however they may call other member operators or function to use their functionality

Binary operators

- Binary operator member functions places an asymmetry on the usage of the operator
  - right hand operand may be converted from another type
  - left hand operand must be of the class type for the operator
- May define global operators functions:

```
inline X operator+( const X& lhsx, const X& rhsx)
{
      return lhsx.add( rhsx);
}
```

Type Conversion Operators

- Type conversion from a user defined type another type can be provided as illustrated:

```
class X{…};

class Y
{
public:
    explicit operator X()
    {
        X temp;
        …
        return temp;
    }
};
```

C++11 introduced the use of explicit keyword for Type Conversion

return type

User Defined Literal

- Many suffixes for literals of built in types
- C++11 introduce the capability to define user define literals (suffix _ as without suffix are reserved)
- Definition uses operator syntax:

```
class Meters
{
  long double _distance;
public:
  Meters(long double km) :_distance(km) {}
};
```

Suffix for literal

```
inline Meters operator"" _km(long double km)
{
return Meters( km *1000);
}
```

Double quotes

```
Meters meters = 2.0_km;
```

Kilometers to meters

Overview of Design Patterns

- The section gives an introduction to many issues relating to the necessity and implementation of Design Patterns:
  – Handle/Body Idiom
  – Bridge Pattern
  – Singleton

Handle/Body Idiom

- The Handle Body Idiom is a commonly used idiom to help with decoupling
- Decoupling has many benefits:
  - Separation to help with team working
  - Help to reduce need for header file inclusion
  - Used in a number of design patterns

Forward Declaration

- The need for header files can be reduced by the use of forward declarations:

```
#include "A.h"

class B
{
        A *pAData;
...
};
```

If only pointer or reference is used the compiler does not need the implementation of A at this point

```
class A;

class B
{
        A *pAData;
...
};
```

Forward Declaration

Forward Declarations

- As the technique of a using forward declaration is so useful the library provides for this
- 'iostream' is a commonly used header file
  - Not needed every if only using references
- Use 'iosfwd' which contains information required by compiler

Handle/Body and Inheritance

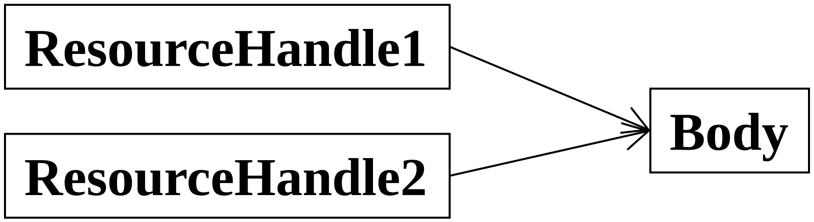
- Inheritance is a standard and commonly used principle of Object Oriented Programming
- However there is a trend to reduce the use of Inheritance, due to:
  - Inheritance is a strong relationship
  - Derived classes depend on base
  - As code is modified over time dependency can become strained
    - Syntactic or Semantic
- Delegation through use of Handle/Body can be more robust

Handle/Body – PIMPL Idiom

- The Handle/Body is used in the PIMPL Idiom
  - PIMPLE – 'Pointer to Implementation'
  - Sometime referred to as the "Cheshire Cat Idiom"
    - Common idioms often have a number of names
- The 'handle' would simple hold a pointer to the 'body'
  - Also allows more sophisticated implementations such as having a shared body
    - Implement by 'reference' counting

Shared Body

- Implementation using shared body can help minimise memory allocation requirements
  - Typically implemented using reference counting



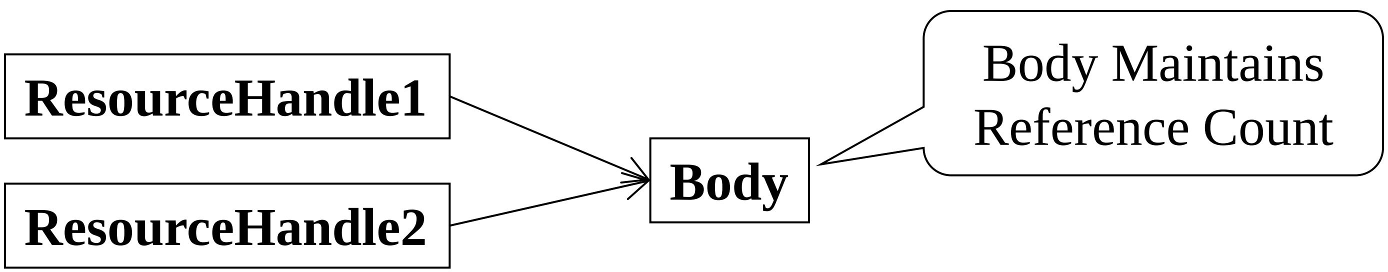
  - Body would only be deleted when last handle deleted
  - Changing one 'handle' requires coping of body (Copy On Write)

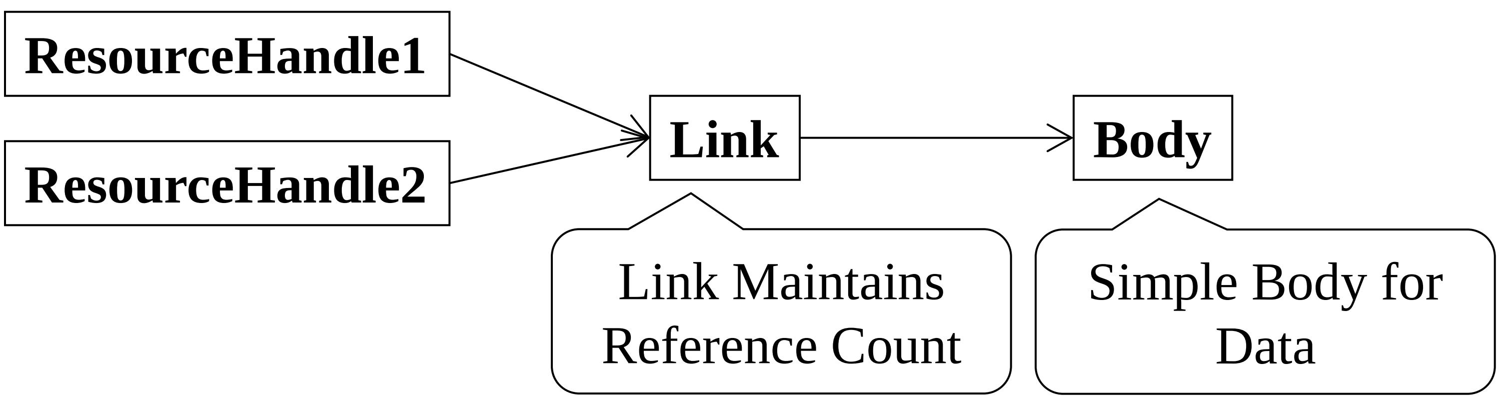
Immutable and Copy On Write

- Immutable types can be useful, especially for 'functional' style programming
- Immutable objects do not change there value
  - Any operation to 'change' the value results in a new object being created
  - Thus the use of Copy On Write to implement
- Immutable type could be implemented as either a simple 'Value Type' or using the PIMPL idiom

Handle/Body Implementations

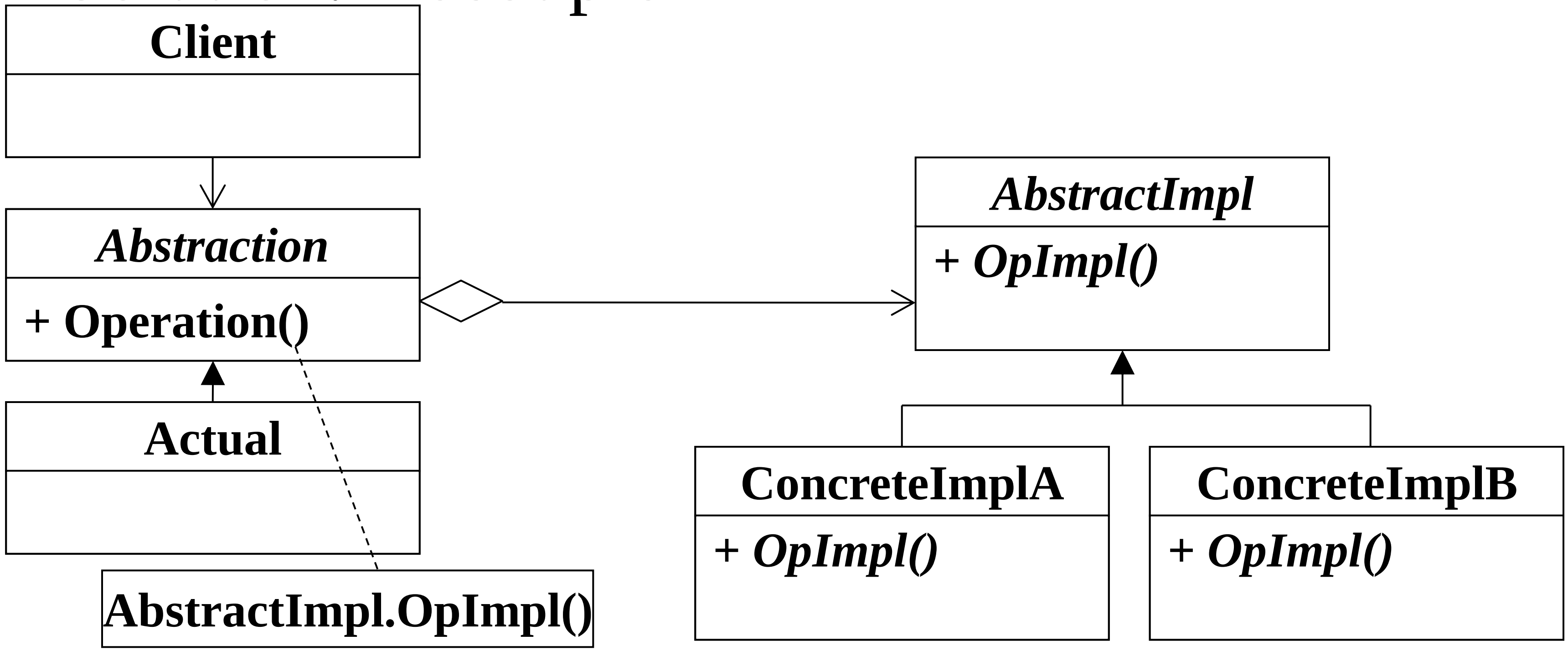
- Where the handle/body idiom is being implemented for shared body there are a number of ways in which this can be implemented:



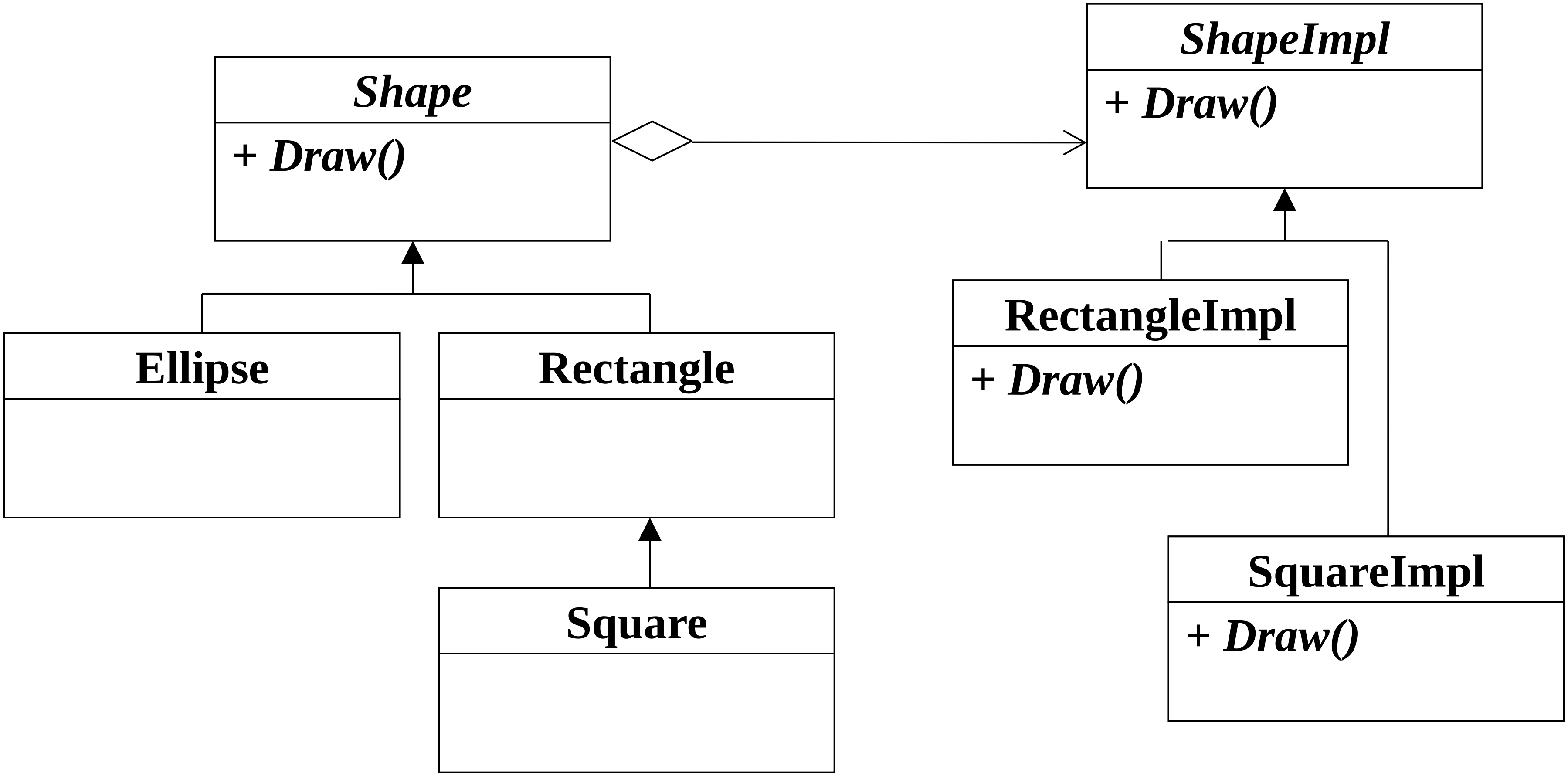


Bridge Pattern

- Problem: Mismatch between abstraction and implementation
- Solution: Decouple



Bridge - Shapes



Smart Pointers

- This Section covers:
  - RAII
  - Smart Pointers
  - std::shared_ptr
  - std::unique_ptr

Resource Acquisition is Initialisation

- Common idiom "Resource Acquisition Is Initialisation" (RAII)
- Classes typically defined to:
  - initialize resource in its constructor
  - tidy up or release resource in destructor
  - provide access to resource, either
    - mimic interface; or
    - possibly use smart pointer

Smart Pointers

- Smart Pointers wrap ordinary pointers
  - Provide automatic freeing of memory
- Standard Library provides 'auto_ptr' (from C++98)

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

...
{
        auto_ptr<int> ptr(new int);

        *ptr = 43;
}
```

Now deprecated in C++11

memory automatically freed when ptr object goes out of scope

std::shared_ptr (C++11)

- There are many implementations of Smart Pointer
  - With different semantics
- C++11 provides 'shared_ptr'
  - Has appropriate semantics for usage within STL

```
#include <memory>
...
{
        std::shared_ptr<int> ptr(new int);
        std::shared_ptr<int> ptr2;
        ptr2 = ptr;

        if(ptr) *ptr = 43;
}
```

Many Constructor Overloads

memory managed by two objects

memory freed when ptr and ptr2 go out of scope

std::unique_ptr

- Only one unique_ptr can manage an object
  - **Not copyable or assignable**

```
std::unique_ptr<int> ptr(new int);

*ptr = 43;
std::unique_ptr<int> ptr2;

ptr2 = std::move(ptr);          Move pointer

// Not valid to use ptr as pointer now moved from 'ptr'

ptr.reset();  // Reset to 0
```


Make Functions

- Make functions can be used to avoid explicit instantiation of objects, uses constructor:

```
class PassengerDetails
{
   std::string _name;
   int _weight = 0;
public:
   PassengerDetails() {}
   PassengerDetails(std::string n, int w) :_name(n), _weight(w) {}
…
};
```

```
std::shared_ptr<PassengerDetails> pd =
                          std::make_shared<PassengerDetails>("Fred", 34);
```

std::weak_ptr

- ‘weak_ptr’ is used to break possible cycles
  - Same pointer as a shared pointer but does not increase reference count until ‘lock’ is called:

```
std::shared_ptr<int> ptr(new int(43));
std::weak_ptr<int> wp = ptr;

*ptr = 43;
{
   auto temp = wp.lock();
   if( temp)
   {
      int val = *temp;
   }
}
```

Returns shared_ptr

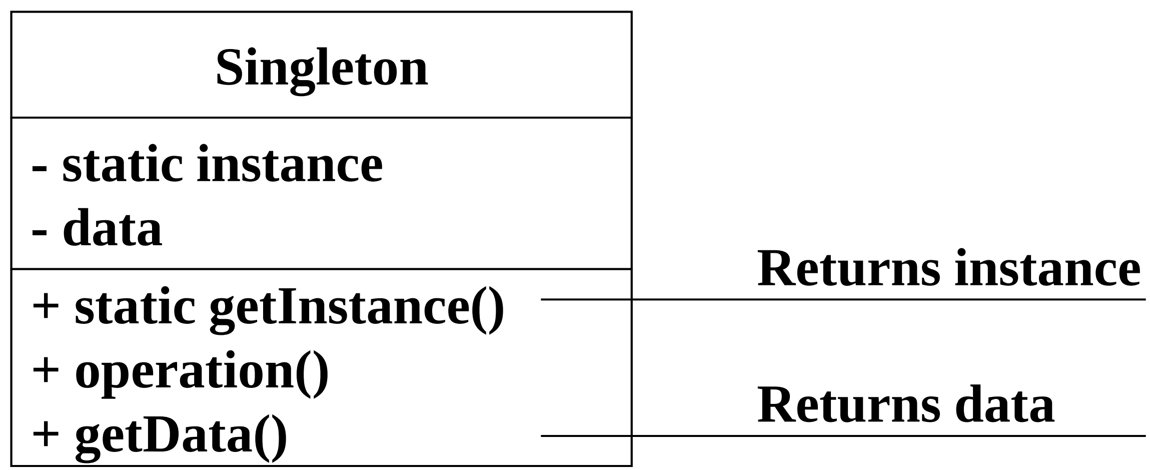
Release ‘lock’.

Singleton

- Problem:
  - Need to access a single object from 'anywhere' within application/project
  - There must only be one instance of this object
- The use of a Singleton is also motivated by the consideration that global data is 'evil'
  - However, some see Singletons as a Global
  - Singletons should be used with caution
- Singletons are controversial!!

Singleton Diagram

- Many Patterns are illustrated as UML diagrams
  – UML diagrams have the advantage of providing a language independent description



Singleton Type?

- There are many motivations for wanting a single instance:
  - In memory state – possibly configuration
  - Façade or Factory (for creation of other objects)
  - Other patterns such as State
  - Universal output – such as Logger

Singleton Problems

- Whilst as a diagram the singleton is one of the simplest it is also one of the most controversial
- In the broader sense how many 'singletons' should be created?
  - One per process/machine/network/country…
  - Physical or Logical 'one'
    - Just one instance or one set of state
      - E.g. controlling access to set of physical ports on a machine?

Singleton Implementation Problems

- Design Patterns are applicable to Object Oriented Design, however language can affect implementation
- Potential problems:
  – What is the lifetime of a singleton?
  – How does singleton behave in multithreaded environment?
  – Will the application scale? (consider previous slide)
  – Should it support inheritance?

Singleton Implementation (Naïve Implementation)

- One possible implementation:

```
class DataSingleton
{
   static DataSingleton *_instance;
   string _data;
   DataSingleton(){}
public:
   static DataSingleton* get_instance()
   {
      if( _instance == nullptr) _instance = new DataSingleton();
      return _instance;
   }
   string get_data() const { return _data; }
};
```

No public constructors

Implemented using Lazy Initialization

```
DataSingleton* instance = nullptr;
```

Singleton Implementation (Magic Static)

- Local static storage is safe for initialisation from multiple threads:

```
class DataSingleton
{
   DataSingleton(){}
public:
   static DataSingleton& get_instance()
   {
      static DataSingleton instance;

      return instance;
   }
};
```

No public constructors

Only initialised once

Static Object

- The use of a static object as a singleton does have precedence of usage in some frameworks!

```
Application the_app;

class Application
{
  Application() {…}
public:
  static Application *get_app()
  {
    return &the_app;
  }
};
```

Meyers Singleton

- Template Singleton:

```
template<typename T> class TheSingleton
{
public:
  static T& instance()
  {
    static T the_instance;
    return the_instance;
  }
  virtual ~TheSingleton(){}
};
```

```
class ActualSingleton :
    public TheSingleton<ActualSingleton>
{
  friend class TheSingleton<ActualSingleton>;
protected:
  ActualSingleton(){ …}
};
```

No public constructors

Singleton Lifetime

- Allow deleting of singleton object:

```
class TempSingleton
{
  TempSingleton(){}
  static TempSingleton* _instance;
  static void free() {  delete _instance;
                        _instance = nullptr; }
  friend class LiveTemp;
public:
  static TempSingleton *create()
  {
    if( !_instance) _instance = new TempSingleton();
    return _instance;
  }
};
```

Free object

```
class LiveTemp
{
  public:
  ~LiveTemp()
  {
    TempSingleton::free();
  }
};

LiveTemp aLive;
```

Lifetime controlled by scope of object

Variations on Singleton

- There are many variations on the Singleton Pattern
  - The implementation of the Singleton on the previous is not thread safe
    - If two threads attempt to get the instance, for the first time, it is possible that these thread get distinct objects
    - A number of variations resolve this problem
      - Either, protect access to the creation of the Singleton
      - Or create the object in advance (i.e. do not use Lazy Initialization)

Alternative to Singleton

- Singletons have multiple responsible, both being a container for data and responsible for object creation
  - Better to separate responsibilities
- Due to the controversial nature of the Singleton there are alternative approaches
  - Use factories to create objects
    - Control creation in another way
  - Use Mono-state Pattern
    - Define an ordinary class, but with a static field for the data
    - Provide public method/property to access static field

Mono-State Pattern

- One possible implementation:

```
class MonoState
{
  static Data *_instance;
public:
  MonoState(){}
  Data* get_data()
  {
    return _instance;
  }
};
```

Inherit from abstract class to allow dependency injection

- Many objects of this type can be created but all access the same underlying data

Standard Library

- Standard Library Features
- Containers
- Algorithms
- Function Objects
- Lambda Expressions
- C++11 new Types

Standard Library Features

- The Standard Library contains:
  - string
  - complex
  - streams
  - Standard Template Library (STL)
    - Container classes - list, vector, map, etc.
    - Iterators - to traverse containers and used within algorithms
    - Algorithms - to act on containers via iterators
    - Function objects
- Library is fairly low level!

Containers

- The STL provides a number of template container types.
- Storage within the containers puts some requirements on the stored object types (typically):
  - default constructor
  - destructor
  - assignment operator
- The containers and algorithms are designed to be efficient.
- Containers:
  - sequential (deque, list, vector)
  - associative (map, multimap, set, multiset)
  - adaptors (stack, queue, priority_queue)
- A number of containers are guaranteed to be contiguous
  - vector, string, array and valarray

list

- Doubly linked list of items:

```
#include <list>

void do_work()
{
   std::list<int> cntrInts;

   for( int i = 0; i < 100; i += 2)
   {
      cntrInts.push_back( i);
      cntrInts.push_front( i+1);
   }
   cntrInts.sort();
}
```

Declaring a list of integers

Alternately pushing values onto the front and back of the list

Sorting the list uses operator < for integers

Container - emplace

- Emplace methods have been added which allows creation of objects in place within a container:

```
std::vector<int> data;
data.emplace(data.begin(), 3);

data.emplace_back(8);
```

- Emplace function take Rvalue references

Container Traits

- The Standard Library code make use of known types as traits
- Containers provide:

```
typedef typename _thebasee::value_type value_type;
typedef typename _thebasee::size_type size_type;
typedef typename _thebasee::difference_type difference_type;
typedef typename _thebase::pointer pointer;
typedef typename _thebase::const_pointer const_pointer;
typedef typename _thebase::reference reference;
typedef typename _thebase::const_reference const_reference;
```

- Now typically implemented as ‘alias’

Iterators Categories

- Five iterator categories, with increasing flexibility:

Iterator category	**Convention (documentation)**	**Restrictions**	**Operations supported**
Output	OutIt	Single pass	++(increment), *(de-reference)
Input	InIt	Single pass	++(increment), *(de-reference)
Forward	FwdIt	Multipass	++(increment), *(de-reference)
Bi-directional	BiIt	Like forward but can move backwards	++(increment), --(decrement), *(de-reference)
Random access	RndIt	Like pointer	++(increment), --(decrement), *(de-reference), + N (arithmetic)

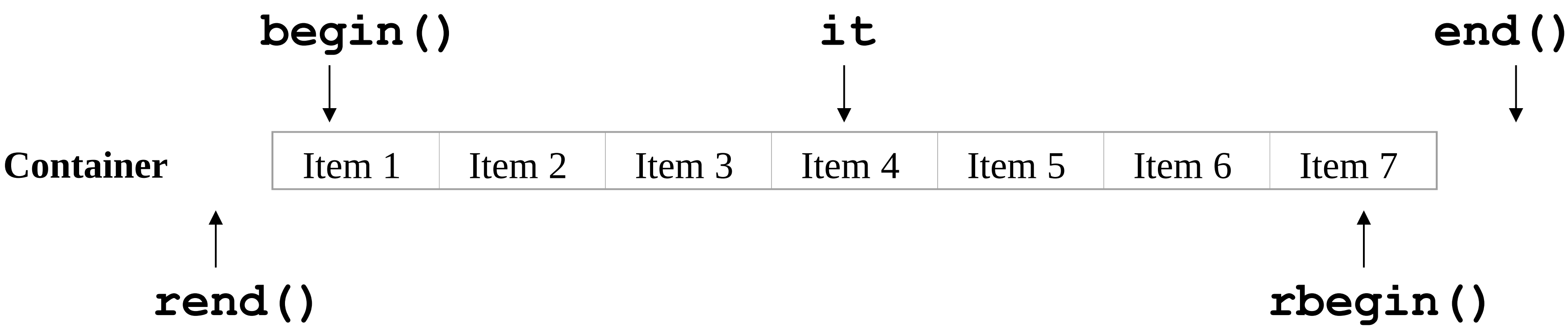
Iterators

- Generalisation of pointers:

```
{
    vector<int> data;
    vector<int>::iterator it = data.begin();

    advance( it, 3);
    // ...
}
```

advance algorithm works on all but output iterators



vector

- Behaves like a variable length array:

```
const int SIZE = 10;
using namespace std;

void doWork()
{
   vector<int> vecInt(SIZE);

   for( int i = 0; i < vecInt.size(); ++i)
        vecInt[i] = i;

   vector<int>::const_iterator it = vecInt.cbegin();

   for( ; it != vecInt.cend(); ++it)
       cout << *it << endl;
}
```

#include <vector> and <iostream>

Initial size

Access like array

Use iterator to traverse container

Typedef Types

- When using Containers many types are defined for use with them
  - The earlier slide illustrates the use of ‘iterator’ and ‘const_iterator’
    - Provide appropriate iterator types for contained data
  - E.g. for vector:

Type	**Description**
iterator	Iterator to elements
const_iterator	Iterator to constant element
reverse_iterator	Reverse iterator to elements
const_reverse_iterator	Const reverse iterator to elements

Iterator Traits

- Iterator Traits provides a unform interface for iterators usage:

```
typedef ptrdiff_t difference_type;
typedef _Ty value_type;
typedef _Ty *pointer;
typedef _Ty& reference;
typedef random_access_iterator_tag iterator_category;
```

- Now typically implemented as ‘alias’

```
using difference_type = ptrdiff_t;
using value_type = _Ty;
using pointer = _Ty *;
using reference = _Ty&;
using iterator_category = random_access_iterator_tag;
```

Vector Usage

- Behaves like a variable length array:

```
using namespace std;
const int SIZE = 10;

void doWork()
{
   vector<int> vecInt;

   vecInt.reserve(SIZE);

   for( int i = 0; i < vecInt.capacity(); ++i)
        vecInt.push_back(i);
   //...
}
```

#include <vector> and <iostream>

Initial capacity

Fill to Capacity!

std::map

- Associative container between two different types:

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

void doWork()
{
   map<string,int> cntrStringValue;

   cntrStringValue["fred"] = 4;
   cntrStringValue["jim"] = 3;

   map<string,int>::const_iterator
                          it(cntrStringValue.cbegin());

   for( ; it != cntrStringValue.cend(); ++it)
      cout << (*it).first << ' ' << (*it).second << endl;
}
```

Also need to include ‘iostream’ and ‘string’

Define key and value types

Define key/value pair

‘indexed’ by string!

Strict Weak Comparable

- Some Containers and Algorithms require implementing operator <:

```
class Data
{
        int _i;
public:
        Data( int i):_i(i){}
        bool less_than( const Data& d) const
        {
                return this->_i < d._i;
        }
};
bool operator<( const Data& lhs, const Data& rhs)
{
        return lhs.less_than( rhs);
}
```

Strict Weak Comparable

- When implementing operator <, this should be done in a way similar to the way < between int or double is implemented, i.e.
  - Strict
    - X < X is false
  - Weak
    - X < Y is false and Y < X is false implies X and Y equivalent!
  - Ordering
    - X < Y is true and Y < Z is true implies X < Z is true

Algorithms

- Many algorithms exist:
  - copy, for_each, find, find_if, count, sort, transform, etc.
- The purpose of some is self evident from the name
  - The 'if' algorithms take a predicate (method returning boolean)
- Algorithms work on containers via iterators or use pointers
- Many algorithm allow the application of functions, function objects or lambda expressions

algorithms

- Many new algorithms have been added (including):

Algorithm	**Description**
all_of	Test condition for all in range satisfying condition
any_of	Test condition for any in range satisfying condition
none_of	Test condition for none in range satisfying condition
find_if_not	Find element if not satisfying predicate
is_permutation	Check if ranges are permutations
copy_n	Copy number of elements
copy_if	Copy if elements satisfy predicate
copy	Copy range of elements
move	Move range of elements
shuffle	Rearrange elements in range randomly
is_sorted	Check if elements in range is sorted

STL Algorithm copy

- Algorithms will work with iterators or pointers:

```
#include <algorithm>

int main()
{
   int arr[] = { 32, 43, 65, 654, 743};
   int arr2[10];

   std::copy( arr, arr+sizeof(arr)/sizeof(int), arr2);

   std::copy( std::begin(arr), std::end(arr), arr2);

   return 0;
}
```

five values copied to arr2

C++11 functions works with arrays and containers

Using Iterators

- The range to be iterated across is identified by the use of begin and end iterators:

```
std::copy(data.cbegin(), data.cend(), dataCopy.begin());
```

  - Use const iterators where possible (and appropriate)
- If there is the possibility the functionality will be used within a template function, use:

```
std::copy(std::cbegin(data), std::cend(data), std::begin(dataCopy));
```

std::cbegin and std::cend (C++14)

string – Container of char?

- string may be accessed via iterators

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

int main()
{
	std::string s( "Fred bloggs");
	std::ostream_iterator<char> osit( cout, " ");
	std::copy( s.cbegin(), s.cend(), osit);

	return 0;
}
```

Using const iterators into string!

Algorithm sort

- Work with pointers and arrays

```
#include <algorithm>

int main()
{
   char chArr[] = "The quick brown fox";
   std::sort( chArr, chArr + sizeof(chArr) - 1);
   // characters sorted into alphabetic order

   return 0;
}
```

- Also work with STL containers via iterators

ostream_iterator

- ostream_iterator< > is a special purpose output iterator.
- Used to output general types to an ostream
  - operator << (put to) must be defined.

```
#include <iterator>        // required for ostream_iterator
#include <algorithm>       // required for copy
#include <array>

//...
{
   std::array<int, 7> arrayInt { 3, 5, 6, 76, 2, 54, 9};
   std::ostream_iterator<int> osi( std::cout, " ");

   std::copy( arrayInt.cbegin(), arrayInt.cend(), osi);
   std::cout << std::endl;
}
```

C++11 would require {{ …}}

Initialised with output stream and delimiter

istream_iterator

- ostream_iterator< > is a special purpose input iterator.
- Used to output general types to an ostream
  - operator >> (get from) must be defined.

```
#include <iterator>          // required for ostream_iterator
#include <algorithm>         // required for copy
#include <vector>
#include <iostream>

//...
{
   vector<int> vecInt;
   std::istream_iterator<int> isi( std::cin);

   std::copy( isi, istream_iterator<int>(),
                              back_inserter( vecInt));
}
```

Numeric Algorithms

- Within the header file <numeric> are a number of additional algorithms
  - Not the least of which is 'accumulate'
  - Often described as allowing summation of a range!
- Accumulate applies a function/functor to each element, but threading a variable through

```
struct sum
{
        auto operator()(int a, int b) const -> int{ return a+b;}
};
int main() {
        std::vector<int> data{ 34, 34, 65,5,345, 6};
        int answer = std::accumulate(data.cbegin(), data.cend(), 0, sum());
}
```

Initial value

Removing Items

- Removing items from a container creates an interesting challenge, as it is important not to invalidate iterators
  - There are a number of options for removing items
  - Removing by moving values up does not change the size of the container
  - Remove copy populates a new container with the remaining items

std::remove and std::remove_if

- std::remove and std::remove_if remove items by moving values in the container
  - The container size remains the same
  - Returned value is new ‘end’ iterator
    - Can use ‘erase’ member function to get rid of items!

```
void remove_value(std::vector<int>& data, int value)
{
   auto end = std::remove(std::begin(data), std::end(data), value);

   auto newEnd = std::remove_if(std::begin(data), std::end(data),
                              std::bind(std::less<int>(),std::placeholders::_1, 3));
}
```

Compared using
== operator

std::remove_copy

- std::remove_copy provides one way to copy all items except specific ones:

```
void remove_value(const std::vector<int>& data, int value)
{
   std::vector<int> result;

   std::remove_copy(std::cbegin(data), std::cend(data),
                    std::back_inserter(result), value);

}
```

Results copied to new container

Compared using == operator

- There is also is std::remove_copy_if

Function Objects

- Objects of a type which implements the operator (); the function operator.
- STL provides:
  - less< >, less_equal< >, greater< >,…
  - plus< >, minus< >,…
  - also, base type
    - binary_function< >
- Advantage:
  - inlining of operator() more efficient than function call

Function Object (Functor)

- Want to count values greater than 10:

```
struct gt_10                Define function object type?
{
   bool operator()( int a) const
   {
      return a > 10;           Number hard coded!
   }
};
size_t count_gt_10( vector<int>& vInt)
{
   return std::count_if( vInt.cbegin(),
                         vInt.cend(), gt_10());
}
```

Transform and Functor

- Transform can take function or Functor:

```
struct cube
{
  int operator()(int a) const { return a*a*a;}
};
int main()
{
  std::vector<int> vecInt;

  for(int i = 0; i < 10; ++i) { vecInt.push_back(i);}
  std::transform( vecInt.begin(), vecInt.end(),
      vecInt.begin(), cube());

  return 0;
}
```

Populate Vector

Apply Functor to Values

Function Object Adaptor

- Want to count values greater than some value:

```
#include <functional>

size_t count_gt_n( vector<int>& vecInt, int n)
{
   return std::count_if( vecInt.cbegin(),
                         vecInt.cend(),
                         std::bind2nd( std::greater<int>(), n));
```

Deprecated in C++11

- Many predefined function objects (types!):
  - greater, less, greater_equal, less_equal

std::bind

- 'bind1st' and 'bind2nd' are now superseded by the bind method from similar to that in 'boost' library:

```
int compose( int a, int b, int c)
{
        return a*100 + b * 10 + c;
}
```

Three parameter function

Bind second argument to first parameter

Bind first argument to second parameter

```
auto fun = std::bind( compose, std::placeholders::_2, std::placeholders::_1,3);

int result = fun(1, 2);
```

Pass 2 to a and 1 to b

Result: 213

std:bind

- Want to count values greater than some value:

```
#include <functional>

size_t count_gt_n( const std::vector<int>& vecInt,
                                                int n)
{
   return std::count_if( vecInt.cbegin(),
                         vecInt.cend(),
                         std::bind(
   std::greater<int>(),
   std::placeholders::_1,
   n));
}
```

Bind first argument to first

Lambda Expressions (C++11)

- C++11 had introduced Lambda Expressions
  - Alternative to Named Function Objects
- Below is an illustration of the usage:

```
#include <functional>

size_t count_gt_5( const std::vector<int>& vecInt)
{
   return std::count_if( vecInt.cbegin(),
      vecInt.cend(),
      [] (int a) -> bool { return a > 5;});
}
```

Lambda Introducer

Value hard coded in

Lambda Syntax Options!

- Lambda expressions can typically be written in a number of ways:

```
auto lambda1  = [](int n) -> int { return n + n;};

std::cout << lambda1(7) << std::endl;

auto lambda2 = [](int n) { return n + n;};

std::cout << lambda2(8) << std::endl;

auto lambda3 = [] { return 42;};

std::cout << lambda3() << std::endl;
```

Full definition including return type

Return type deduced

No parameters

Closure (C++11)

- C++0x had introduced Lambda Expressions
  - Closures can use values within local scope
- Below is an illustration of the usage:

```
#include <functional>

size_t count_gt_n( vector<int>& vecInt, int n)
{
   return std::count_if( vecInt.cbegin(),
      vecInt.cend(),
      [=] (int a) -> bool { return a > n;});
}
```

Indicates values from outer scope can be read

Lambda Introducer

- The lambda introducer can be used to indicate the use (or modification) of variables from the outer scope:

Symbols/Values	**Description (captures)**
=	Read values in scope
&	Reference variables from in scope
a, b	Read variables a and b in scope (values constant within lambda)
&a, &b	Reference variables a and b in scope
this	Reference to current object defined within an instance member function!

Recursive Lambda Function

- Lambda Functions assigned to a variable
- Recursive Lambda Functions can also be defined, as follows:

Include <functional>

```
std::function<int(int)> fact =
  [&fact] (int a) ->
  int { return a == 0 ? 1 : a*fact( a-1);};
```

for_each Algorithm

- The for_each algorithm can be used to iterate across a collection executing a method for each element:

```
std::for_each( vecInt.begin(),
          vecInt.end(),
          [] (int& a) -> void { a *= 2;});
```

Lambda Expression

For each could use a Function Object, Pointer to Function or Lambda Expression

C++11 new Types

- C++11 has added a number of new types:
  - tuple (define object containing data of different types)
  - ref (reference wrapper allow ‘reference’ to variable by value)
- Containers:
  - Sequencial (array, forward_list)
  - Associative (unordered_set / unordered_multiset and unordered_map / unordered_multimap)
- Regular Expressions - Regex

std::tuple

- Tuple is a new C++11 type which allows simple creation of objects to wrap data
  - Can be used to return multiple values from a function, without the need create a new custom type

Initialise with Constructor

Explicit definition of tuple

```
std::tuple<int,double, std::string> values(7,2.3, "Greetings");

std::cout << std::get<0>(values) << std::endl;
std::cout << std::get<1>(values) << std::endl;
std::cout << std::get<2>(values) << std::endl;
int a;
double d;
std::string msg;
std::tie( a, d, msg) = values;
```

Use 'get' function to access values (zero based index)

Set values on variables

std::make_tuple

- A tuple can be created from values using std::make_tuple function:

Comma separated values to be wrapped in tuple

```
auto values = std::make_tuple( 7, 2.3, "Greetings");

std::cout << std::get<0>( values) << std::endl;
std::cout << std::get<1>( values) << std::endl;
std::cout << std::get<2>( values) << std::endl;
```

Use ‘get’ function to access values (zero based index)

```
auto values1 = std::make_tuple( 7, 2.3, "Greetings");
auto values2 = std::make_tuple( "Hello", 5.5);
auto values = std::tuple_cat( values1, values2);
```

Concatonate tuples into single tuple

std::array

- std::array is a sequential container which is fixed size (at compile time)
  - std::array provides a wrapper for arrays
  - Allows array to behave as standard container

Member function	**Description**
begin()	Iterator to beginning
end()	Iterator end
cbegin()	Constant iterator to beginning
cend()	Constant iterator to end
size()	Size of array
fill()	Fill with a value

std::forward_list

- std::forward_list is a singly linked list allowing it to grow or shrink

Member function	**Description**
begin()	Iterator to beginning
end()	Iterator end
front()	Return first element
push_front()	Push element onto front of list
pop_front()	Remove element from front of list
insert_after	Insert element after iterator
sort	Sort elements

std::unordered_map

- Similar to std::map but does not provide ordering of keys:

```
std::unordered_map<int, std::string> numbers;
numbers[3] = "three";
numbers[5] = "five";
numbers[11] = "Eleven";      Initial setting for 11
numbers[12] = "twelve";

numbers[11] = "eleven";      Value not duplicated
for (auto p : numbers)
{
   std::cout << "Key: " << p.first << ", " << "value: " << p.second << std::endl;
}
```

std::unordered_multimap

- Similar to std::map but does not provide ordering of keys, but allows multiple entries:

```
std::unordered_multimap<int, std::string> intstringValues;
for (int i = 0; i < 10; i++)
{
      intstringValues.insert({ i, "Fred" + std::to_string(i) });
}

intstringValues.insert({ 5, "Fred5" });
intstringValues.insert({ 7, "Fred7" });
intstringValues.insert({ 11, "Fred11" });
intstringValues.insert({ 20, "Fred20" });
```

Convert integer to string

Entries duplicated for keys

Completing Key

- For user defined class ‘MyNumber’ implement ‘hash’ member function and:

```
struct MyNumberHasher
{
    size_t operator()(const MyNumber& mn) const noexcept
    {       return mn.hash();    }
};
inline bool operator==(const MyNumber& lhs, const MyNumber& rhs)
{
    return lhs.getVal() == lhs.getVal();
}
…
    std::unordered_multimap<MyNumber, std::string, MyNumberHasher>
            intstringValues;
…
```

Call hash member function

Use hasher to instantiate multimap

Regular Expression Search

- Regular expression searches can be perform:

```
#include <iostream>
#include <string>
#include <regex>
…
```

String to search

```
std::string msg("The quick brown fox jumps over the lazy dog!");

std::regex reg("brown fox", std::regex_constants::icase);
```

String to search for

Ignore Case

```
if (std::regex_search( msg, reg))
{
   std::cout << "Contains 'brown fox'" << std::endl;
}
```

Regular Expression Search

Regular Expression Iterator

- Regular expression iterators allow traversing matches:

String to search

Regular Expression to Search Find

Iterator to Matches

End Iterator

```
std::string msg("The quick brown fox jumps over the lazy dog!");

std::regex word("\\S+");


auto word_begin = std::sregex_iterator(msg.begin(), msg.end(), word);
auto word_end = std::sregex_iterator();

for (auto it = word_begin; it != word_end; ++it)
{
   std::cout << it->str() << std::endl;
}
```

Boost

- This section gives an overview of the Boost C++ library:
  - Boost Introduction
  - Installing Boost
  - Project Properties
  - Overview of Boost Library
  - boost::tuple
  - boost::any
  - boost::ref
  - Smart pointers
  - boost::bind

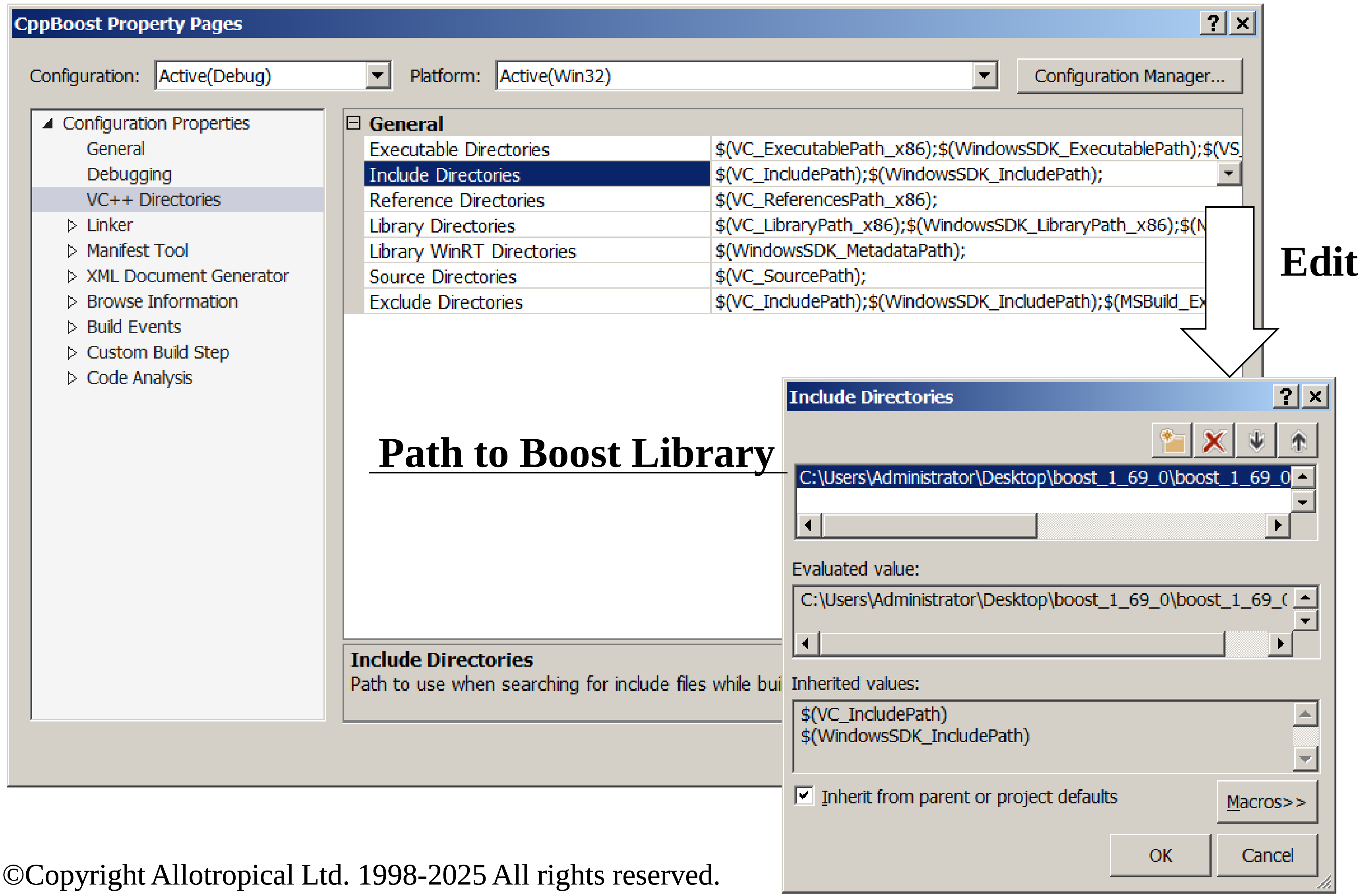
Boost Introduction

- The Boost library is a highly regarded open source C++ library
- Provides a wide range of functionality
- Many similar principle to C++ Standard library
- Many features now being incorporated into the newer C++ standards
- Supports a range of platforms
- Used by many organisations

Installing Boost

- Boost is relatively straightforward to use:
  - Download appropriate compressed file from:
    - https://www.boost.org/
  - Decompress file to a suitable location
  - Within Visual Studio add C++ project
  - Within project properties add a include directory for the decompressed boost folder
  - Within source files add include files for required functionality

Project Properties – Include Directories



Overview of Boost Library

- The boost library is divide into various directories:

Directory	**Features**
algorithm	Various algorithms
atomic	Synchronisation primitives and guards
bind	Bind functionality
container	Containers
filesystem	Cross platform support for file systems
functional	Function wrapper functionality
smart_ptr	Various smart pointers
thread	Threading support

boost::tuple

- 'tuple' provides a type to hold multiple items of data:

```
#include <iostream>
#include <boost\tuple\tuple.hpp>

int main()
{
   boost::tuple<int, double, std::string> values(7, 2.3, "Greetings");
   std::cout << boost::get<0>(values) << std::endl;
   std::cout << boost::get<1>(values) << std::endl;
   std::cout << boost::get<2>(values) << std::endl;

   return 0;
}
```

boost::any

- 'any' provides the capability to wrap any 'value_type', that is, a copy constructible type:

```
#include <iostream>
#include <boost\any.hpp>

int main()
{
  boost::any aval = 23;
  data d(7);
  boost::any ad = d;
  …
```

```
  if (!ad.empty())
  {
    std::cout << ad.type().raw_name() << std::endl;
  }
  std::cout << boost::any_cast<data>(ad).getA() << std::endl;
  ad.clear();
  if (ad.empty())
  {
    std::cout << "Empty!" << std::endl;
  }
  return 0;
}
```

boost::ref

- 'ref' provides a reference wrapper to allow passing by reference to template functions:

```
#include <boost\ref.hpp>

template<typename T> void do_work(T t)
{
   pass_ref(t);
}
void pass_ref(data& d)
{
   d.setA(7);
}
```

```
int main()
{
   data d{8};
   doWork(boost::ref(d));
   std::cout << d.getA() << std::endl;

   return 0;
}
```

shared_ptr/make_shared

- 'shared_ptr' is supported from boost:

```
#include <iostream>
#include <boost\smart_ptr\shared_ptr.hpp>
#include <boost\smart_ptr\make_shared.hpp>

int main()
{
   boost::shared_ptr<int> pInt = boost::make_shared<int>(7);
   std::cout << *pInt << std::endl;

   return 0;
}
```

boost::bind

- 'bind' is available within the 'boost' library:

```
int compose( int a, int b, int c)
{
        return a*100 + b * 10 + c;
}
```

Three parameter function

```
auto fun = boost::bind( compose,
                        boost::placeholders::_2, boost::placeholders::_1,3);

int result = fun(1, 2);
```

Bind second argument to first parameter

Bind first argument to second parameter

Pass 2 to a and 1 to b

Result: 213

Recommended Reading

- The C++ Programming Language
  - by Bjarne Stroustrup
- Effective C++
  - by Scott Meyers
- Exceptional C++
  - by Herb Sutter
- Design Patterns: Elements of Reusable Object-Oriented Software
  - by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides
- Generic Programming and the STL
  - by Matthew Austern

The End

Congratulations on completing this C++ course