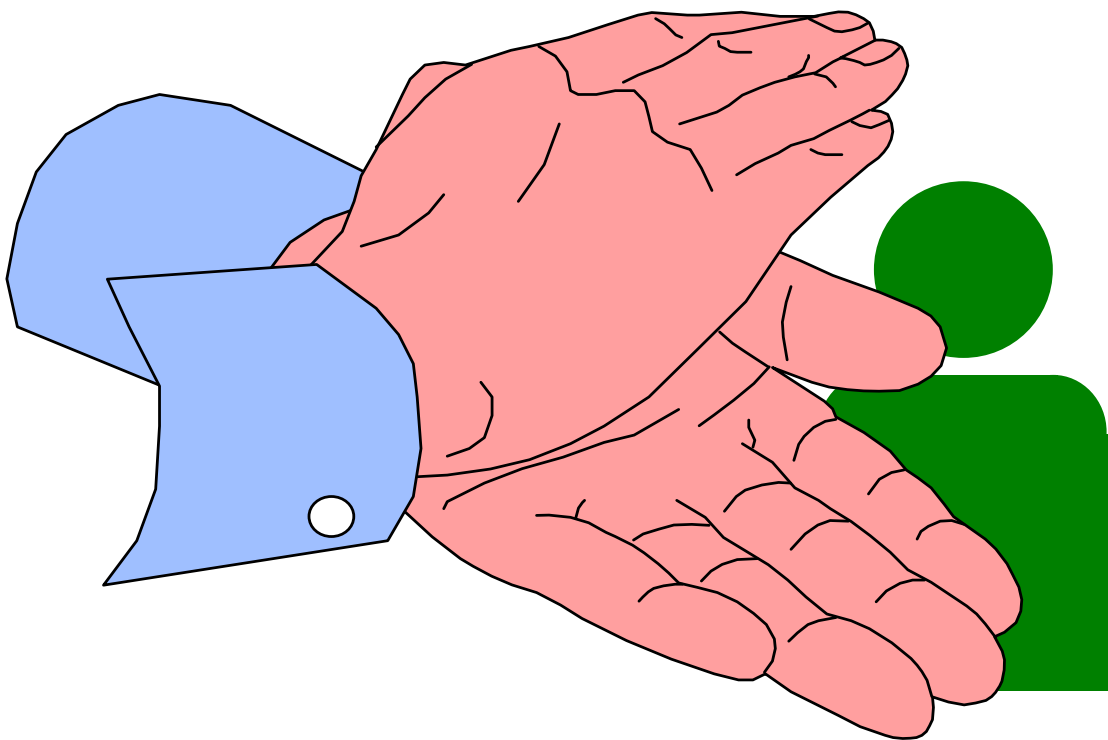


Advanced C++



Chris Seddon

seddon-software@keme.co.uk

Advanced C++

- 1 Casting**
- 2 Namespaces**
- 3 Const Correct**
- 4 Exception Handling**
- 5 Template Functions**
- 6 Template Classes**
- 7 Advanced Templates**
- 8 Handle Body**
- 9 Managed Pointers**
- 10 Boost**
- 11 Object Oriented Callbacks**
- 12 Memory Management**
- 13 Singleton**
- 14 Object Factories**
- 15 Multiple Inheritance**
- 16 C++ 2011**

Chapter 1

1

Copyright ©1994-2010 CRS Enterprises

3

Casting

- The new cast operators
 - `static_cast`
 - `dynamic_cast`
 - `const_cast`
 - `reinterpret_cast`
- Runtime type information
 - `typeid`



Chapter 1

The 4 new Cast Operators

- **static_cast**
 - checks validity of cast at compile time
- **dynamic_cast**
 - checks validity of cast at run time
- **const_cast**
 - overrides const correctness during cast
- **reinterpret_cast**
 - casting between unrelated types

C++ has four new cast operators. These cast operators are designed to split the all embracing conventional cast into manageable categories. Conventional casts are often hard to spot and sometimes generate incorrect code. The new cast operators are intentionally verbose to highlight the cast and lend the full weight of the compiler to resolving problems.

static_cast

1. Polymorphic Casts

compile time check if conversion is legal

2. Non-polymorphic Casts

compile time check of casts between built-in types (int, double, etc)

compile time check of casts between non pointer types

cast operators must be defined for classes concerned

```
A a = static_cast<A>(b)
```

`static_cast` can perform conversions between pointers to related classes, not only from the derived class to its base, but also from a base class to its derived. This ensures that at least the classes are compatible if the proper object is converted, but no safety check is performed during runtime to check if the object being converted is in fact a full object of the destination type. Therefore, it is up to the programmer to ensure that the conversion is safe. On the other side, the overhead of the type-safety checks of `dynamic_cast` is avoided.

static_cast

- Non polymorphic casts

```
int main()
{
    BBB b;

    // Static casting BBB objects
    pa = static_cast<AAA*> (&b);
    pb = static_cast<BBB*> (&b);
    pc = static_cast<CCC*> (&b);
}
```

```
class AAA
{
public:
    virtual void fa1() { cout << "fa1" << endl; }
    virtual void fa2() { cout << "fa2" << endl; }
    virtual void fa3() { cout << "fa3" << endl; }
};

class BBB
{
public:
    virtual void fa1() { cout << "fa1" << endl; }
    virtual void fa2() { cout << "fa2" << endl; }
    virtual void fa3() { cout << "fa3" << endl; }
};

class CCC
{
public:
    virtual void fa1() { cout << "fa1" << endl; }
    virtual void fa2() { cout << "fa2" << endl; }
    virtual void fa3() { cout << "fa3" << endl; }
};
```

Copyright ©1994-2010 CRS Enterprises

7

Static casts were introduced to C++ to resolve some of the problems associated with conventional casts. For example consider the three conventional casts above

```
AAA* pa = (AAA*) &b;
BBB* pb = (BBB*) &b;
CCC* pc = (CCC*) &b;
```

All three casts compile, but the first and third class are clearly wrong. These casts actually set the pointers to the address of b and then when the pointers are dereferenced

```
pa->fa2();
pb->fb2();
pc->fc2();
```

all three statements use the V-Table for BBB to determine the function to be called. Thus in all three cases the second function in the BBB's V-Table is called, namely fb2(). The functions fa2() and fc2() are never called. In fact if the parameters of these functions are different, the two incorrect calls will throw exceptions. Clearly counter intuitive!

If you use static casts, the compiler picks up the erroneous casts early

```
pa = static_cast<AAA*> (&b);
pc = static_cast<CCC*> (&b);
```

and generates compiler errors. Now, there is no chance of calling the wrong function through the V-Table.

reinterpret_cast

Allows any integral type to be converted into pointer type and vice versa

Always between Non Polymorphic Types

intent is to highlight cast between two apparently unrelated classes
may indicate a kludge or temporary fix

```
struct io_port
{
    unsigned char status;
    unsigned char data;
};

volatile io_port *iop = reinterpret_cast<io_port*>(0xF878);
```

pointer type

integral type

Copyright ©1994-2010 CRS Enterprises

8

The **reinterpret_cast** operator also allows any integral type to be converted into any pointer type and vice versa. Misuse of the **reinterpret_cast** operator can easily be unsafe. Unless the desired conversion is inherently low-level, you should use one of the other cast operators.

The **reinterpret_cast** operator can be used for conversions such as `char*` to `int*`, which is inherently unsafe.

The result of a **reinterpret_cast** cannot safely be used for anything other than being cast back to its original type. Other uses are, at best, nonportable.

The **reinterpret_cast** operator converts a null pointer value to the null pointer value of the destination type.

dynamic_cast ...

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

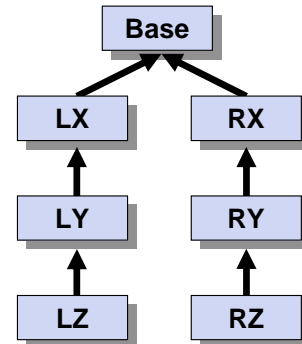
struct Base { virtual void f() {} };

struct LX : public Base { virtual void lf() {} };
struct LY : public LX { virtual void lf() {} };
struct LZ : public LY { virtual void lf() {} };

struct RX : public Base { virtual void rf() {} };
struct RY : public RX { virtual void rf() {} };
struct RZ : public RY { virtual void rf() {} };
```

- **cast performed at runtime**

- null pointer returned on failure (pointers)
- or exception thrown (references)



Copyright ©1994-2010 CRS Enterprises

9

Consider the above inheritance hierarchy. If we create objects in this hierarchy, some casting operations will make sense while others will not. For example, can we cast to an LX class? Obviously only objects in classes derived from LX (i.e. LX, LY and LZ) satisfy this condition.

But how can we check for valid casts to LX at runtime? That's where the dynamic cast comes into its own. The dynamic cast will succeed for valid casts, but will return a null pointer if the cast is invalid.

... dynamic_cast

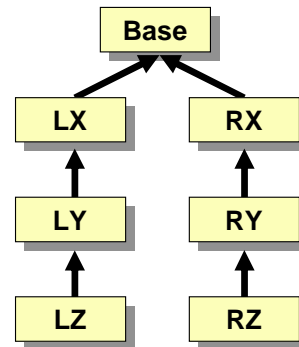
```
int main()
{
    vector<Base*> theList;

    theList.push_back(new Base);
    theList.push_back(new LX);
    theList.push_back(new LY);
    theList.push_back(new LZ);
    theList.push_back(new RX);
    theList.push_back(new RY);
    theList.push_back(new RZ);

    unsigned listSize = theList.size();
    for(unsigned i = 0; i < listSize; i++)
    {
        LX* p = dynamic_cast<LX*>(theList[i]);
        if(p) p->lf();
    }
}
```

Copyright ©1994-2010 CRS Enterprises

- **LX, LY, LZ**
 - IS A LX*
 - cast succeeds
- **Base, RX, RY, RZ**
 - null pointer returned



10

Suppose we create a polymorphic collection of Base objects. According to the class hierarchy diagram, the objects in the collection can belong to any of the 7 classes (Base, LX, LY, LZ, RX, RY and RZ). As we iterate through the collection, we can use the dynamic cast to pick out objects that satisfy

object IS LX*

When this program is executed, the dynamic cast succeeds for the LX, LY and LZ objects, but a null pointer is returned for Base, RX, RY and RZ objects.

Dynamic casts can be applied to pointers and references

```
dynamic_cast<LX*>(ptr);
```

```
dynamic_cast<LX&>(ref);
```

const_cast

Removes the const attribute of an object

can also be applied to volatile

```
void print(char * str)
{
    cout << str << endl;
}

int main ()
{
    const char *c = "sample text";
    print(const_cast<char *>(c));
}
```

This type of casting removes the const (or volatile) attribute of an object.

This cast is normally only used with a function that has been written incorrectly such that it requires a non constant argument, but should require a const argument.

Runtime Type Information

- **Limited run information can be retrieved**
 - only the class name can be retrieved
- **Information attached to the V-Tables**
 - not available if no virtual methods
 - but a virtual destructor will suffice

Runtime type information for a class can be retrieved using the `typeid` keyword. The type information is somewhat minimal, basically just the class name.

Note that all runtime type information is always appended to the V-Table, but the V-Table is only generated if there is at least one virtual function in the class. That's why we have added a virtual destructor to the class.

Runtime Type Information

```
using namespace RTTI;
```

```
void DetermineClass(const Person& p)
{
    cout << typeid(p).name() << endl;
}
```

```
int main()
{
    Person p;
    Employee e;
    Salesman s;

    DetermineClass(p);
    DetermineClass(e);
    DetermineClass(s);
}
```

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

```
namespace RTTI
```

```
{
```

```
    class Person
```

```
    {
```

```
    public:
```

```
        virtual ~Person() {}
```

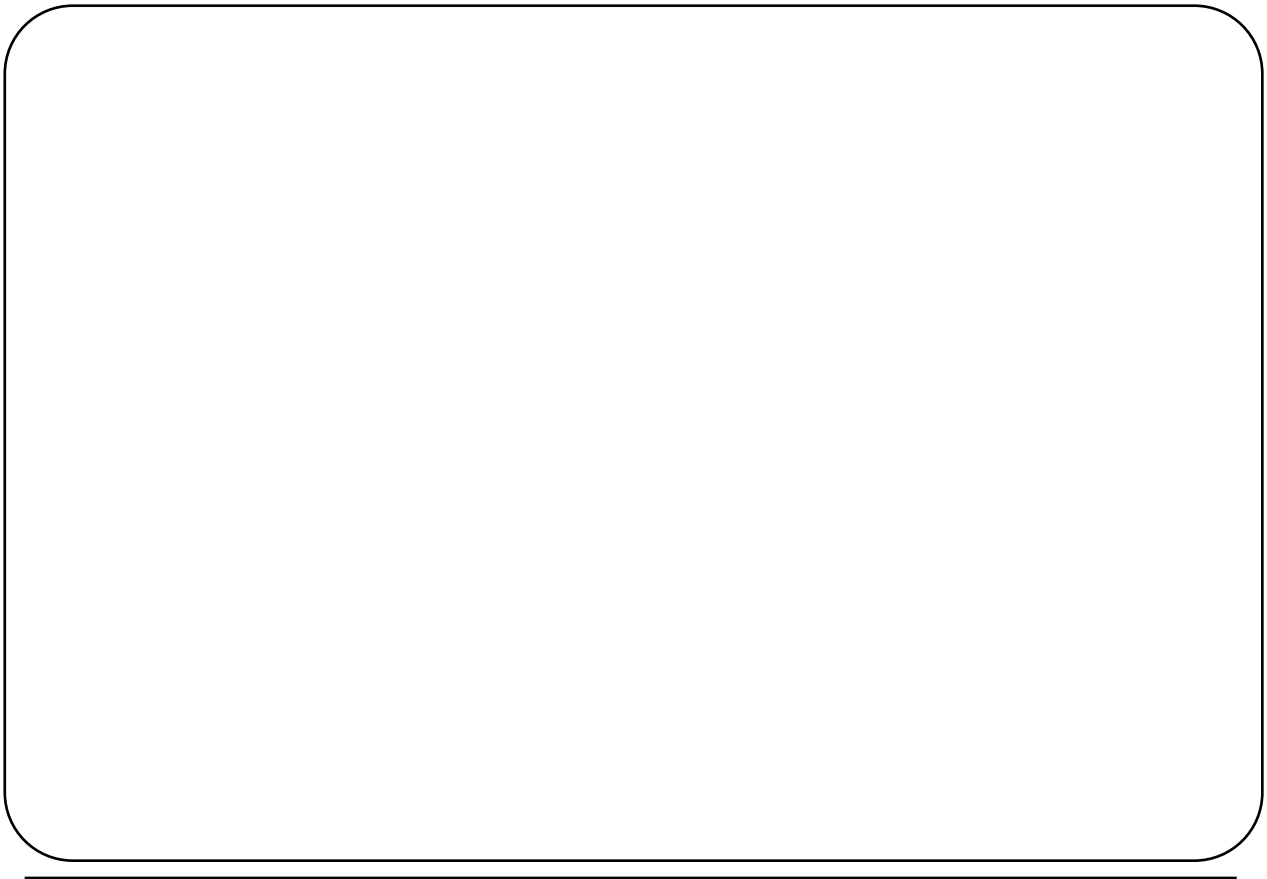
```
};
```

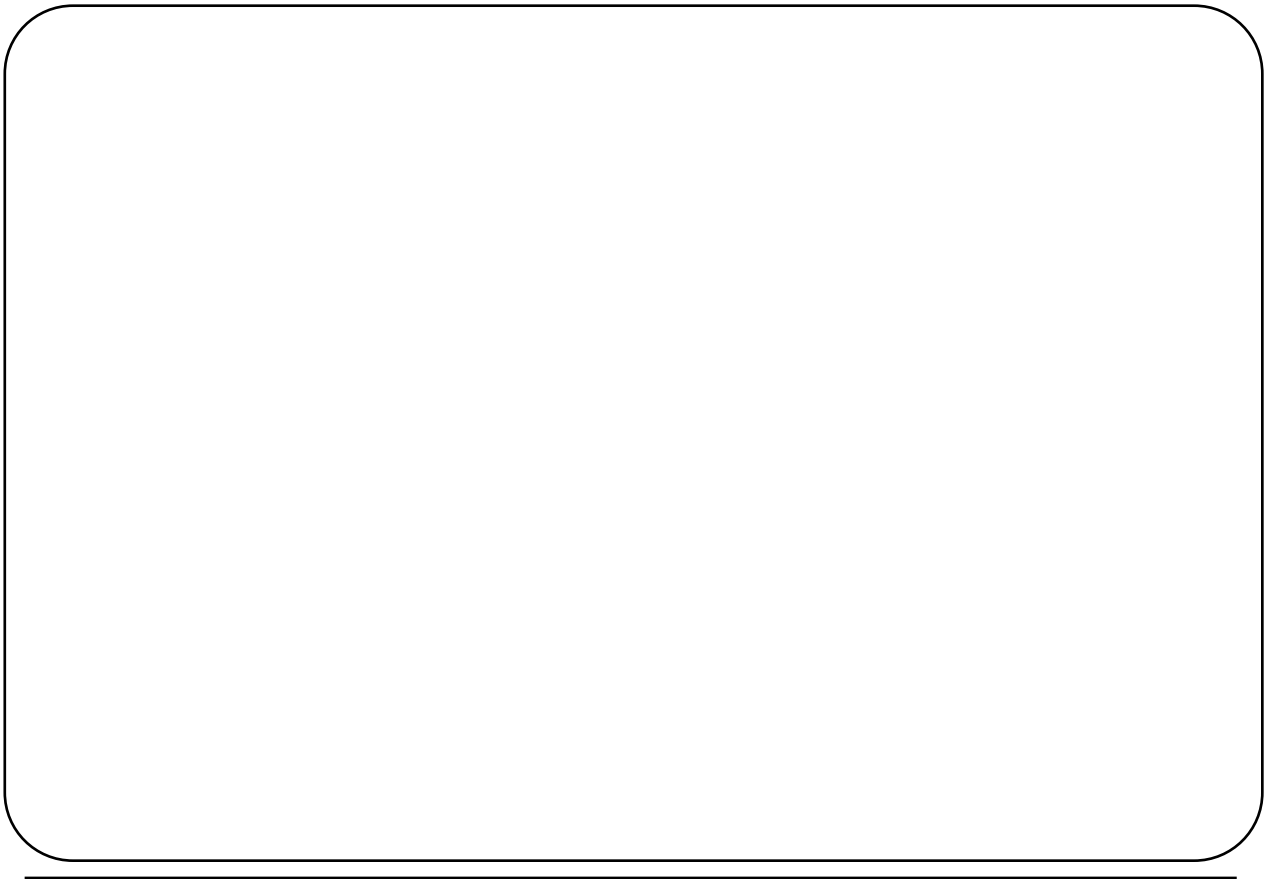
```
    class Employee : public Person {};
```

```
    class Salesman : public Employee {};
```

```
}
```

```
N4RTTI6PersonE
N4RTTI8EmployeeE
N4RTTI8SalesmanE
```





Chapter 2

2

Namespaces

- **Declaring Namespaces**
 - namespace directive
- **Accessing namespaces**
 - fully qualified lookups
 - *using namespace* directive
 - *using* directive
 - namespaces spanning libraries
- **Koenig Lookup**
 - namespaces not independent



Chapter 2

Namespaces

- **Namespaces resolve naming ambiguities**
 - class Map in two different namespaces are resolved as
 - **Geography::Map**
 - **Maths::Map**

```
namespace Geography
{
    class Map
    {
    public:
        void Draw() {}
    };

    class List
    {
    public:
        void Insert() {}
    };
}
```

```
namespace Maths
{
    class Map
    {
    public:
        void Clear() {}
    };

    class List
    {
    public:
        void Insert() {}
    };
}
```

Copyright ©1994-2010 CRS Enterprises

18

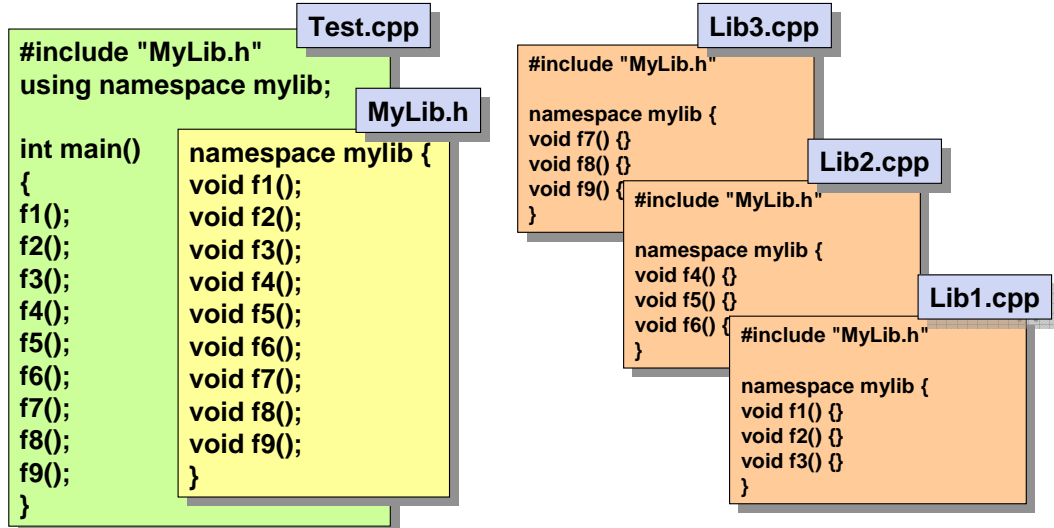
Namespaces were introduced into C++ to resolve naming ambiguities. Consider the two Map classes above. By defining the classes in different namespaces we can differentiate the two classes by

Geography::Map

Maths::Map

Defining a Namespace

- Namespaces can span multiple files
 - but can have several namespaces in the same file



Copyright ©1994-2010 CRS Enterprises

19

Namespaces are often used to group together libraries of related functions and classes. Therefore, it is essential that a namespace is allowed to span multiple source and header files. However, for maximum flexibility, you do have the option of restricting a namespace to a single file, or even grouping several namespaces together in the same file.

using namespace Directive

```
using namespace Geography;
```

```
Map theWorld;  
Maths::Map theNumbers;
```

```
theWorld.Draw();  
theNumbers.Clear();
```

```
namespace Geography  
{  
    class Map { Draw() }  
    class List { Insert() }  
}  
  
namespace Maths  
{  
    class Map { Clear }  
    class List { Insert }  
}
```

- **using namespace**
 - allows unqualified name references
 - can be applied to more than one namespace
 - doesn't stop ambiguities
- **references to other namespaces must be fully qualified**

Copyright ©1994-2010 CRS Enterprises

20

The using namespace directive allows you to use unqualified name references, thereby making your program much less verbose. It is important to realise that the directive can be applied to more than one namespace, so that unqualified names can be used from either namespace. Of course this could reintroduce the naming ambiguities that namespaces were developed to combat.

References to identifiers in other namespaces must be fully qualified

using Directive

```
namespace Geography
{
    class Map { Draw() }
    class List { Insert() }
}

namespace Maths
{
    class Map { Clear }
    class List { Insert }
}
```

```
using namespace Geography;
using Maths::List;
```

```
Map theWorld;
Maths::Map theNumbers;
Geography::List theCountries;
List theSequences;
```

- **using namespace**
 - allows unqualified name references
- **using**
 - resolves ambiguities on name between rival namespaces

unqualified references to Geography
use Maths::List to resolve ambiguities

defaults to Geography::Map
explicit
must be explicit
defaults to Maths::List

The using namespace directive allows unqualified access to identifiers in the namespace, but several using namespace directives in the same program could easily lead to name clashes. To help resolve naming ambiguities without resorting to fully qualified names, you can declare a using directive, such as

```
using Maths::List
```

The using directive becomes the tie breaker if several candidates are found in competing namespaces.

Namespace Aliases

- **namespace keyword**

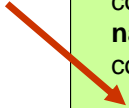
- sets up an alias

oii = outer::inner::in

```
namespace outer {
    namespace inner {
        namespace in {
            int x = 110;
        }
    }
}

int main()
{
    cout << outer::inner::in::x << endl;
    namespace oii = outer::inner::in;
    cout << oii::x << endl;

    using namespace oii;
    cout << x << endl;
}
```



Copyright ©1994-2010 CRS Enterprises

22

Although namespaces are essential for resolving possible naming ambiguities, they can lead to some very verbose code. Fortunately, the namespace keyword can be used to set up an alias for any fully qualified namespace.

In the above example

```
namespace oii = outer::inner::in;
```

sets up a namespace alias so that you can reference x with either of

```
outer::inner::in::x
```

```
oii::x
```

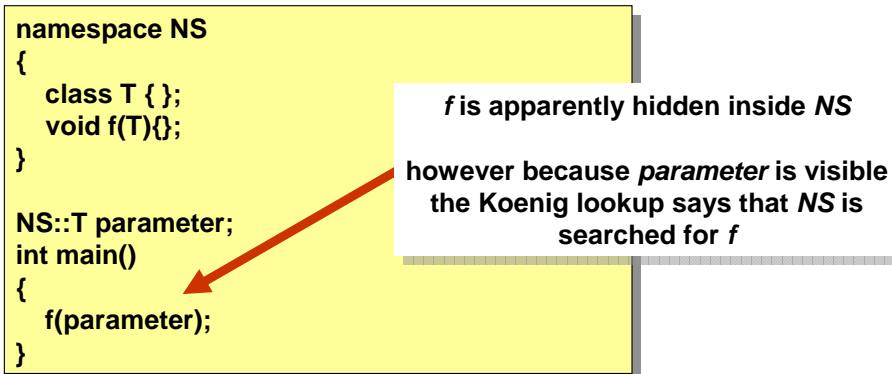
The alias can further be used in

```
using namespace oii;
```

so that x can be referenced without qualification.

Koenig Name Lookup

- When an unqualified name is used in a function call
 - other namespaces not usually considered may be searched



Copyright ©1994-2010 CRS Enterprises

23

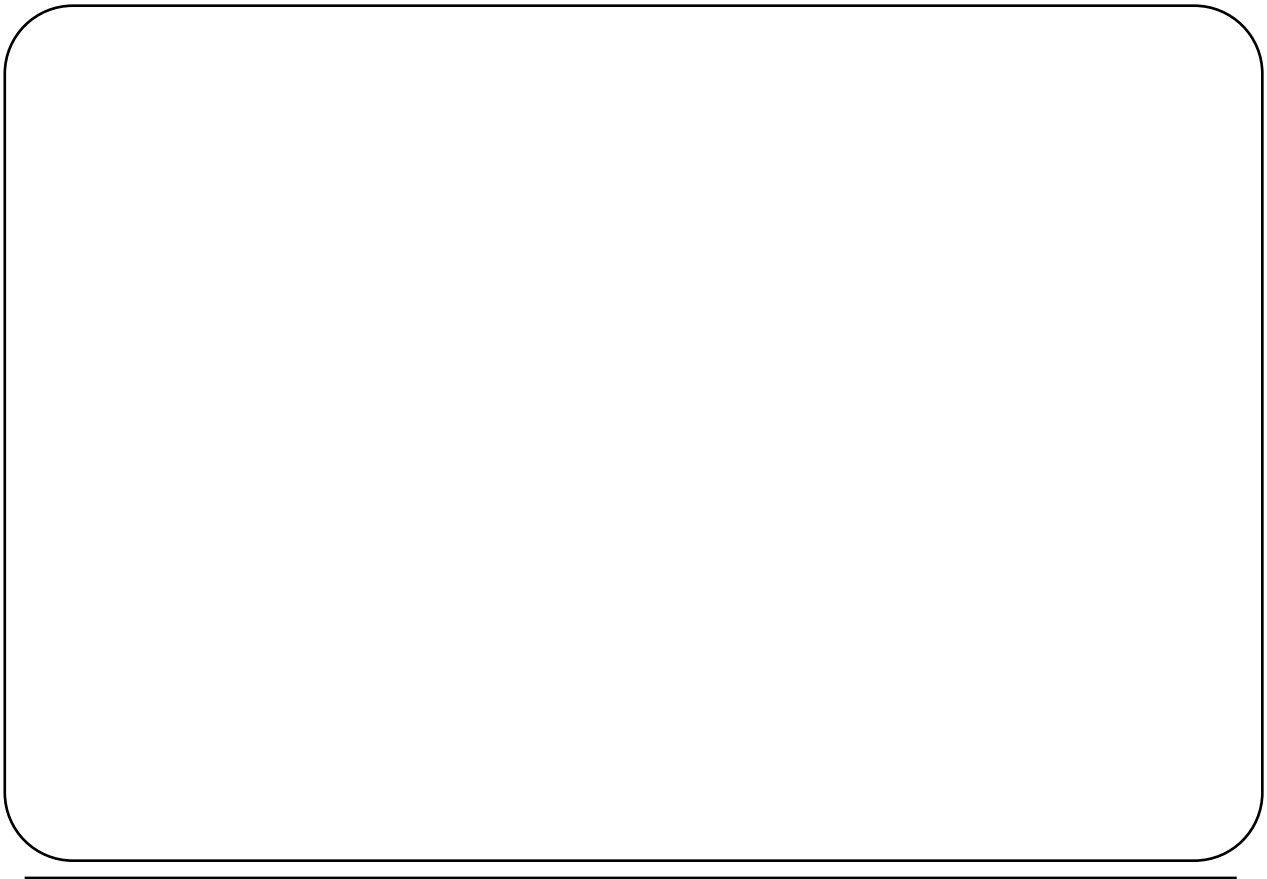
Koenig name lookup is an important part of C++'s scoping rules. It permits sensible access to names defined in a namespace that may not have been explicitly requested with the using directive.

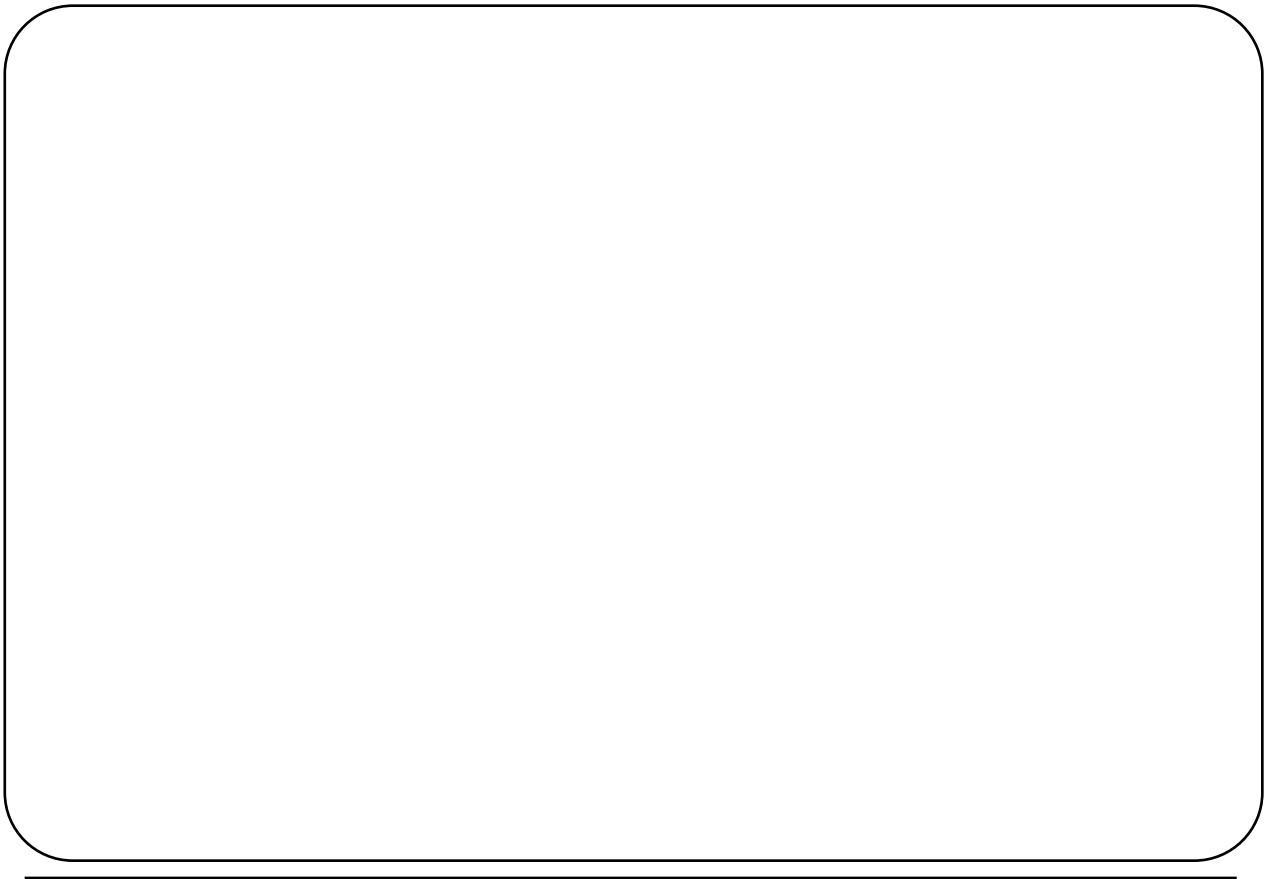
Essentially, the Koenig lookup states that if any of the parameters in a method or function call have been introduced to another namespace (including the default namespace), then that namespace will be searched when selecting candidates for other parameters or even the function name itself.

In the above example, although `parameter` belongs to the `NS` namespace, by declaring it as a global variable in the default namespace, it means that the `NS` namespace is searched in the call

`f(parameter)`

The function `NS:f` is a candidate for `f`. Indeed it is the only candidate for `f`.





Chapter 3

3

Const Correct

- **const functions**
 - used for const objects
- **mutable**
 - distinguish between logical and physical const
- **explicit**
 - to avoid unwanted conversions



Chapter 3

Const is Part of Function Signature

- **const objects can be treated separately**
 - can use a different function
 - add *const* to function signature

```
class Date
{
private:
    int day;
    int month;
    int year;
public:
    Date(int, int, int) {};
    void Print() {}
    void Print() const {}
};
```

```
int main()
{
    const Date christmas(25, 12, 1960);
    Date today(11, 3, 2006);

    christmas.Print();
    today.Print();
}
```

C++ allows class designers to provide different behaviour for constant and non-constant objects. If you define two methods differing only in signature by the *const* keyword, the compiler will ensure *const* objects use the *const* method and non-*const* objects use the non-*const* method.

If only one non-*const* method is provided, it will be used for both *const* and non-*const* objects.

mutable Keyword

- use mutable
 - to distinguish between logical and physical const

```
int main()
{
    const Point p(100, 200);

    p.Print();
    p.Print();
    p.Print();
    p.Print();
}
```

const function

not part of logical const object

```
class Point
{
public:
    Point(int x0, int y0) :
        x(x0), y(y0), calls(0) {}

    void Print() const
    {
        cout << calls << ": Point is at "
            << x << ", " << y << endl;
        calls++;
    }

private:
    int x;
    int y;
    mutable int calls;
};
```

Copyright ©1994-2010 CRS Enterprises

29

Sometimes when you design a class it is helpful to distinguish between logical and physical constness.

For example, the Point class shown above defines three attributes, x, y and calls; x and y are clearly an integral part of any Point object, but in some senses, the calls attribute might be considered as a lightweight attribute, perhaps just added for tracing, profiling or debugging.

Using the mutable keyword, we can mark an attribute as not part of the logical definition of the object, but still part of the physical object. Then if we declare a const Point object any const member function is allowed to modify any mutable attributes - they are not considered as being part of the logical object.

explicit Keyword

- use explicit
 - to avoid unwanted conversions
 - in single parameter constructors
 - cast required
 - to compile successfully

```
int main()
{
    Time t;
    int x = 150;

    t = x;
    t = (Time) x
}
```

*implicit conversion
not allowed*

OK

```
class Time
{
public:
    Time() :
        hours (0),
        minutes(0)
    {}

    explicit Time(int m) :
        hours(m/60),
        minutes(m%60)
    {}
private:
    int hours;
    int minutes;
};
```

Copyright ©1994-2010 CRS Enterprises

30

Operator overloading is an extremely useful part of C++. About the only drawback is when a single parameter constructor is used as a cast operator in situations where no cast is intended

```
t = x;
```

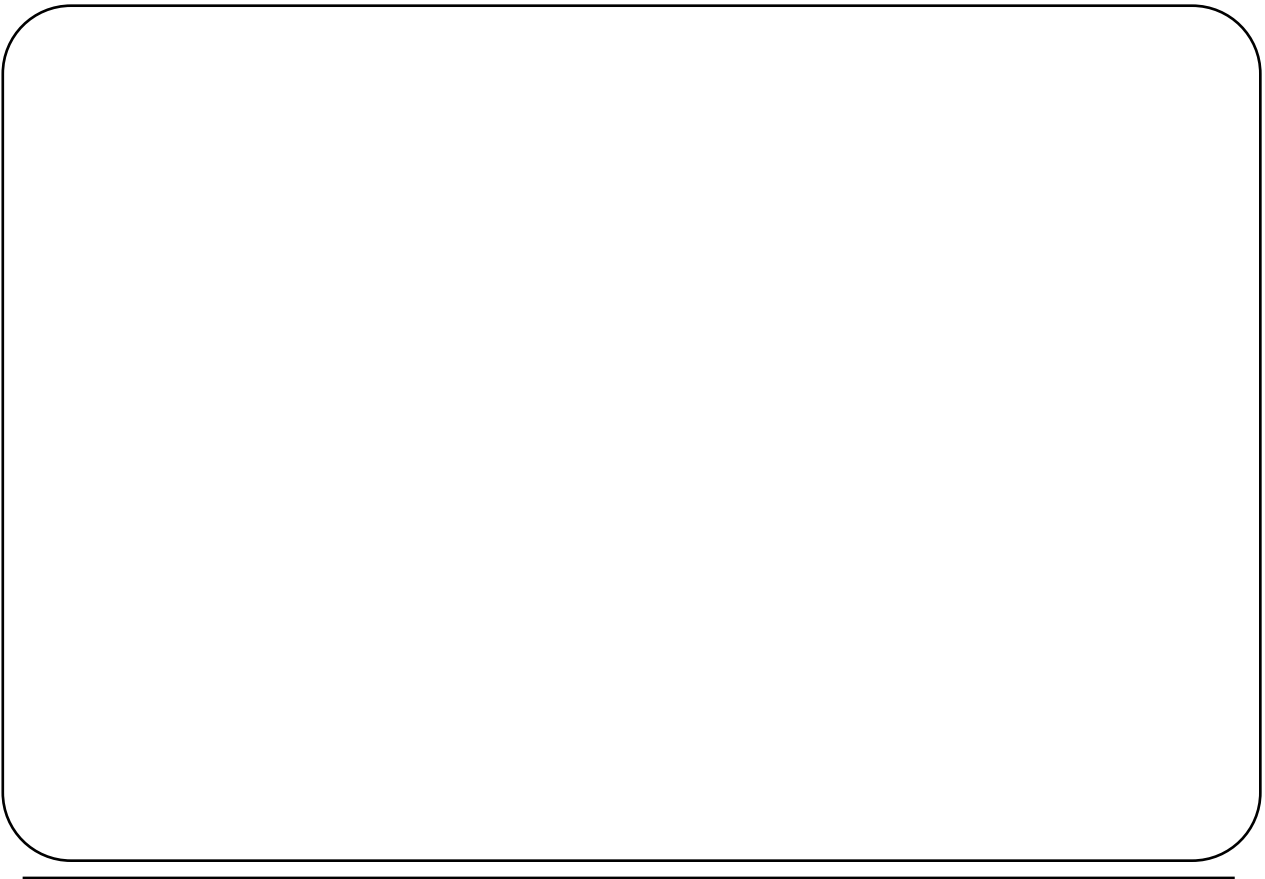
Such conversions are called implicit conversions.

To avoid the above situation, you can adorn a conversion operator with the explicit keyword. The compiler will then only apply the conversion when the client supplies an explicit cast, as in

```
t = (Time) x;
```

volatile Keyword

- ???



Chapter 4

4

Exception Handling

- **Defining Exceptions**
 - Standard Exception Hierarchy
- **Grouping Exception Types**
 - in classes
 - in namespaces
 - in inheritance hierarchies
- **Special Cases**
 - heap based objects
 - exceptions in constructors
 - uncaught exceptions
- **Throw Lists**



Chapter 4

Exceptions

- **Separate error detection from error correction**
 - detection often made in library code
 - library author can detect error, but doesn't know how to correct
 - correction made in user code
 - user can't detect error, but does know how to correct error
- **try**
 - to mark a block as possible source of error
 - try blocks can be nested
- **throw**
 - when error is detected
- **catch**
 - to provide recovery

Copyright ©1994-2010 CRS Enterprises

35

All too often, in traditional C programming, normal code is cluttered with error handling code and the two cannot easily be decoupled. The principal idea behind C++'s exception handling capability is to provide separation of error detection from error correction.

Errors can occur anywhere in a program, but typically errors occur in deeply nested routines such as library modules. When these errors occur, they are usually obvious to the code author, but error recovery is a different matter. Library authors can detect errors, but do not know what the caller wants to do to recover from the error. The caller on the other hand is in the opposite position. The caller finds it difficult to detect errors, but knows what to do if one occurs.

With exception handling separation of responsibilities and decoupling of error detection and correction is easily achieved.

try blocks are used to mark a block of code as a possible source of error. Conceptionally, the try block includes all nested method calls.

throw is used when an error is detected.

catch is used to define code used for error recovery.

Simple Example

```
int main()
{
    Array a;
    int index;
    while(true)
    {
        try
        {
            cout << "Enter array index: ";
            cin >> index;
            cout << a[index] << endl;
        }
        catch (char* errorMessage)
        {
            cout << errorMessage << endl;
        }
    }
}
```

```
class Array
{
private:
    int array[10];
public:
    Array();
    int operator[ ] (int);
};

int Array::operator[ ] (int index)
{
    if (index < 0) throw "index underflow";
    if (index >= 10) throw "index overflow";

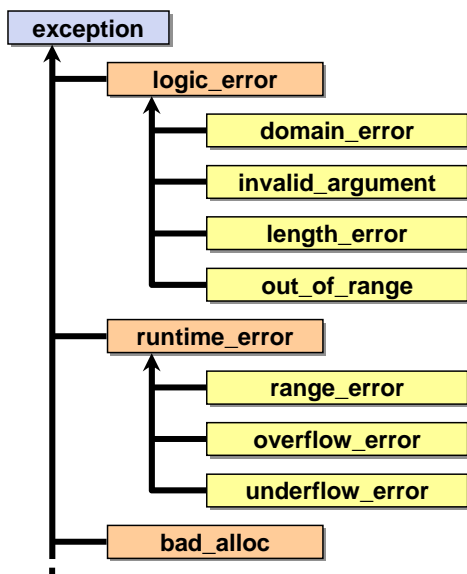
    return array[index];
}
...
```

Copyright ©1994-2010 CRS Enterprises

36

In this simple example an error could potentially occur anywhere in the try block, but in practice the error will occur when the index for the array wrapper object goes out of bounds. The library author (of class array) can detect the array bounds error and throw an exception. The exception is caught in the main program.

The Standard Exception Hierarchy



- **recommendation**

- use the standard exception classes
- or inherit from these classes
 - don't write your own

Copyright ©1994-2010 CRS Enterprises

37

Although you can write your own exception classes it is recommended that you use (or inherit from) the standard exception classes from the standard library:

`exception` is the base class for exceptions

`runtime_error` is the base class for general errors in program execution

`logic_error` is the base class for exceptions that represent a fault in the program's logic, such as a violated function precondition

`runtime_error` is the base class for general errors in program execution

`bad_alloc` is the exception thrown when an attempt to allocate an object using `new` fails

Grouping Exception Types

- **Part of a namespace**
 - exceptions common to a library can be defined within its namespace
- **Part of a class**
 - nest within the class
- **Relating by inheritance**
 - define a hierarchy of exception types

```
namespace utilities
{
    class stack
    {
    public:
        struct error {};
        struct empty : error {};
        void pop();
        //...
    };
}
```

```
class Error { ... }
class MathError : public Error
{
public:
    virtual void Diagnose() {}
};
```

It is not recommended to throw primitive values, such as strings and integers; class based objects can carry more information about the exception and do not rely on the use of error codes.

If an exception is used exclusively by a class, the exception type should be nested within the scope of this class. Similarly if an exception is used frequently throughout a library then it should be made part of the library's namespace.

Furthermore, exceptions may also be grouped using inheritance, with specialisations (e.g. MathsError or RangeError from Error) deriving from a common base class.

Exceptions Using Inheritance

```
int main()
{
    try
    {
        // code that throws MathError
    }
    catch(Error& e)
    {
        e.Diagnose();
    }
}
```

```
class Error
{
public:
    virtual void Diagnose() { ... }
};

class MathError : public Error
{
public:
    virtual void Diagnose() { ... }
};

class FileError : public Error
{
public:
    virtual void Diagnose() { ... }
};
```

- **relate exceptions**
 - using inheritance
- **use polymorphism**
 - to distinguish exceptions
 - e.g. **Diagnose()** method

Copyright ©1994-2010 CRS Enterprises

39

When you define an exception hierarchy it is normal practice to write catch handlers for each of the derived classes. However, with a large hierarchy this will give rise to an untidy list of catch handlers at the end of each try block. An alternative, well worth considering, is to provide a catch handler for the base class and use polymorphism to direct the error to the correct derived class.


In the above example, although we have only provided a catch handler for Error, if a MathError is thrown, then the polymorphic call

```
e.Diagnose()
```

will direct error handling to the MathError class.

Function Exceptions

```
int main()
{
    int result;
    result = f(100);
    result = f(200);
    result = f(-77);
}
```



```
int f(int x)
try
{
    if(x < 0) throw range_error("must be >= 0");
    return x;
}
catch(range_error& e)
{
    cout << e.what() << endl;
    return 0;
}
```

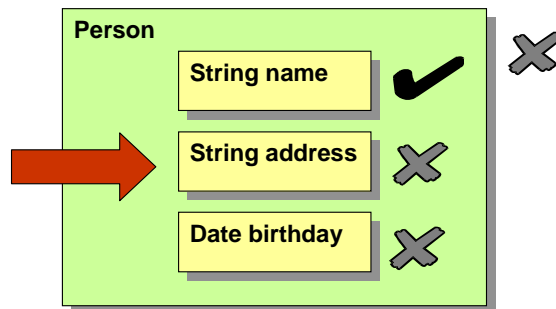
- **wrap an entire function in a try block**
 - separate normal and exception flows
- **variables declared in body ...**
 - not visible in handler
 - unless passes in the exception object

An exception thrown within the function may be handled by catch blocks following the function body. Variables introduced in the function scope are not visible to the catch handlers.

Exceptions in Constructors

- If a constructor throws an exception
 - object is marked as non initialised
 - destructor not called
 - memory reclaimed
- Composite objects clean up fully initialised components
 - destructor is called
 - memory reclaimed
 - other components follow rule above

*exception thrown
in constructor*



Copyright ©1994-2010 CRS Enterprises

41

If an exception is thrown inside the constructor of a stack based object, the object is marked as non initialised. In such cases it doesn't make sense for the object's destructor to be called, but its memory is automatically reclaimed by the compiler.

Composite objects are more complicated. If an exception is thrown inside the constructor of a component object then the above rule applies to the component, the composite object itself and any components not yet initialised. Any components already initialised will be marked as initialised and will be cleaned up (destructor called and memory reclaimed) when the component object goes out of scope at the end of the catch handler.

Problems with Heap Based Objects

- **stack based objects**
 - cleaned up automatically
- **heap based objects**
 - do not get cleaned up automatically
 - alternatively use smart pointers

```
void f2()
{
    Person hilda("Hilda");
    throw "some kind of problem";
}

void f1()
{
    Person* pSteve = new Person("Steve");
    f2();
    delete pSteve;
}
```

*not cleaned up
automatically*



Copyright ©1994-2010 CRS Enterprises

42

The compiler ensures that all stack based objects are cleaned up automatically. However, the compiler is powerless to clean up objects created with the new operator at run time. Hence heap based objects do not get cleaned up automatically.

Cleaning up heap based objects is problematical; it appears that hand crafted code is required. However a generalised method of cleaning up heap based objects can be formulated using smart pointers. This will be addressed elsewhere in the course.

Uncaught Exceptions

- **uncaught exceptions**
 - propagate out of *main*...
- **the *terminate* function is called**
 - calls *abort* by default
 - use *set_terminate*
 - to register an alternative callback
- **use catch all**
 - *catch(...)*
 - must be the last handler in the block

```
int main()
{
    set_terminate(myHandler);
    ...
}
```

```
int main()
{
    try
    {
        ...
    }
    catch(...)
    {
        cerr << "uncaught" << endl;
    }
    ...
}
```

Copyright ©1994-2010 CRS Enterprises

43

If an exception is thrown and there is no associated handler, the exception propagates out of the main program and the C++ standard library function `terminate` will be called. This in turn calls the `abort` function and the program will end abnormally.

You can customise this behaviour by using `set_terminate` to register an alternative handler to be called in the event of an unhandled exception.

Alternatively, you can use the catch all syntax

```
catch(...)
```

to catch an unhandled exception. This catch clause must be the last handler for the try block.

Throw Lists

- Specify which exceptions are thrown

```

class stack
{
public:
    ...
    void pop() throw(empty, bad_alloc);
    void clear();
    size_t size() const throw();
    //...
};

```

Annotations:

- may throw empty or bad_alloc* (points to `pop()`)
- may throw anything* (points to `clear()`)
- throws nothing* (points to `size()`)

- Exceptions are checked against the *throw* list at runtime
 - exceptions not listed generate a call to *unexpected*
 - unexpected* calls *terminate*
 - unless alternative callback is registered using *set_unexpected*

Copyright ©1994-2010 CRS Enterprises

44

You can specify the list of exceptions that may be thrown on a method invocation. The throw specification is part of a function's enforceable contract.

The throw list for `stack::pop` states that it will only generate `empty` or `bad_alloc` exceptions, or exceptions derived from these types.

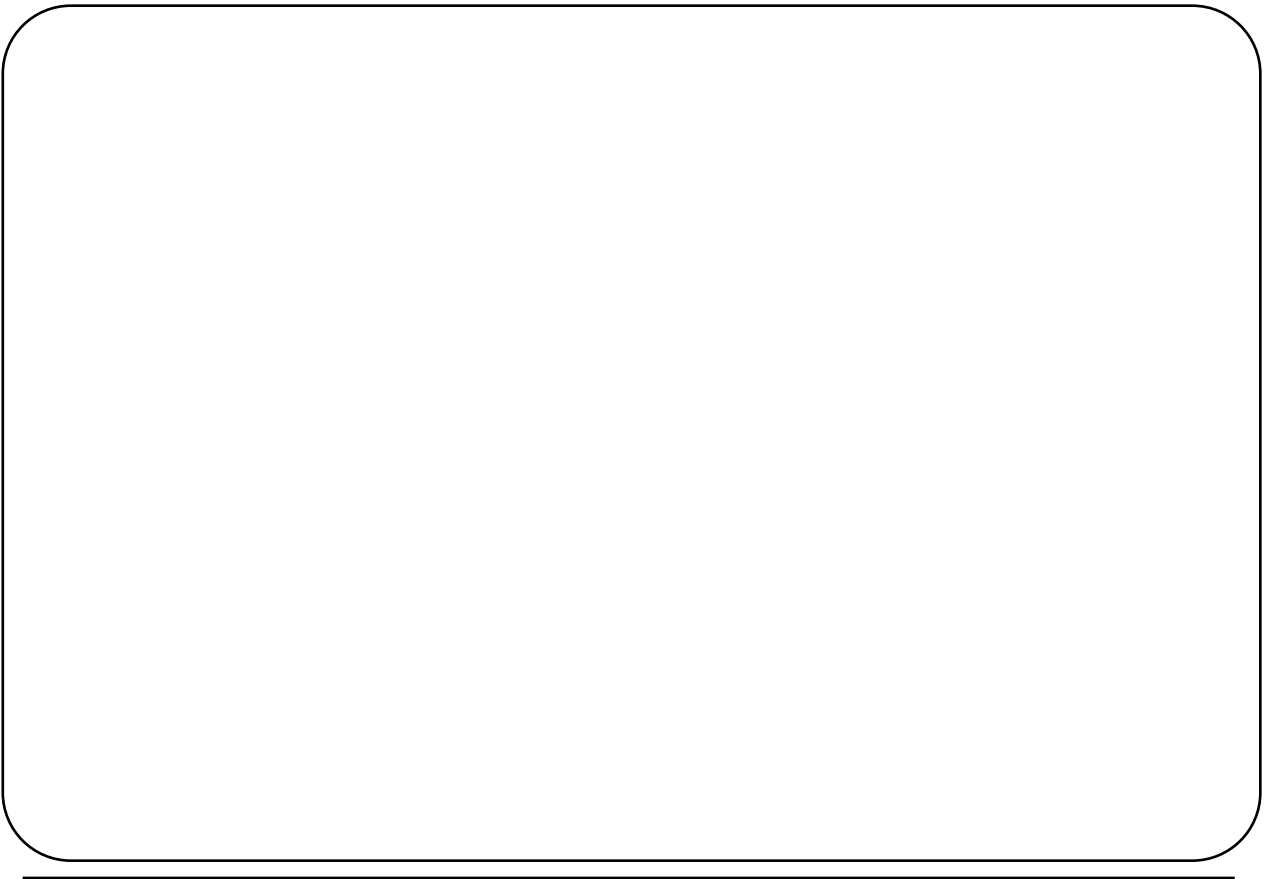
The absence of a throw list, as in `stack::clear` implies that any exception may be thrown.

To indicate that no exceptions are thrown an empty list must be used, as with `stack::size`.

The throw list is not enforced by the compiler; transgressions are checked and handled at runtime. If a method throws any exception not listed in its exception specification, the standard C++ function `unexpected` is called. The default behaviour for `unexpected` is to call `terminate`. Alternatively, you may register a callback using

```
set_unexpected(myHandler)
```

It should be mentioned that many programmers do not approve of the throw list mechanism, mainly because throw lists are not enforced by the compiler.



Chapter 5

5

Template Functions

- **Generic Functions**
 - parameterized types
- **Template Functions**
 - single parameter functions
 - multiple parameter functions
 - ambiguities
 - explicit notation
- **Specialization**
 - exact parameter matches
 - type conversion



Chapter 5

Generic Functions

- **similar functions**
 - ready to be converted to templates
- **identify types**
 - that can be made generic

```
int main()
{
    cout << Max(5, 8) << endl;
    cout << Max(5.1, 8.3) << endl;
    cout << Max("Red", "Blue") << endl;
}
```

Copyright ©1994-2010 CRS Enterprises

```
int Max(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}

double Max(double a, double b)
{
    if (a > b)
        return a;
    else
        return b;
}

string Max(string a, string b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

Consider the three global functions above. Each function performs the identical task of calculating the maximum of two types (int, double or string). Clearly, these functions differ only in terms of the data types used. The code for each function is identical.

Such functions are ideal candidates for being rewritten as a single parameterised template function.

Template Functions

- Parameters can be classes or built in types
 - use **class** or **typename**

```
template <typename T>
T Max(T a, T b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

```
int main()
{
    string s1 = "Red";
    string s2 = "Blue";
    cout << Max(5, 8) << endl;
    cout << Max(5.1, 8.3) << endl;
    cout << Max(s1, s2) << endl;
}
```

calls template with *T = int*
 calls template with *T = double*
 calls template with *T = string*

Copyright ©1994-2010 CRS Enterprises

49

The parameterized function template is written above. Note the use of the template keyword to introduce the function. The template parameters are enclosed in angled brackets (to distinguish them from function parameters).

Templates are instantiated by calling the template function and omitting the angled brackets. The call

Max(5, 8)

will instantiate the template for ints, whereas

Max(5.1, 8.3)

Max(s1, s2)

will instantiate the template for doubles and strings respectively.

Note that convention demands you use a single upper case letter for the template parameter.

Ambiguities

- **Compiler must be able to uniquely determine parameter types**
 - does T = *int* or *double*?
- **These ambiguities rarely arise**
 - types too similar

```
template <typename T>
T Max(T a, T b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

```
double x = Max(5.1, 8);
```

```
double x = Max(8, 5.1);
```

Copyright ©1994-2010 CRS Enterprises

50

With a template function, the compiler must be able to uniquely determine each parameter type from the template call. For example, in the function call

Max(5.1,8)

the template parameter T could be either an int or a double. Ambiguities like this are not allowed and the template call will fail. Fortunately, when templates are used with classes such ambiguities rarely arise.

Multiple Template Parameters

- **Templates can have many parameters**
 - compiler must be able to deduce types

```
template <typename T1, typename T2>
double Max(T1 a, T2 b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

```
double x = Max(5.1, 8);
double y = Max(8, 5.1);
```

T1=double, T2=int

T1=int, T2=double

Templates can have many parameters. In this example the template assumes there are two different types being passed to Max. In all cases the compiler must be able to deduce these types otherwise the template call will fail.

Although from a syntactic viewpoint the above code compiles and runs without a problem, you should consider the semantics of the code carefully. In this example we are finding the larger of two different types and although this makes sense for an int-double combination, does it make sense for doubles and strings?

Ambiguous Return Types

- Return value not considered deterministic
 - compiler can't deduce RET

```
template <typename L, typename R, typename RET>  
RET Max(L a, R b)  
{  
    if (a > b)  
        return a;  
    else  
        return b;  
}
```

```
double x = Max(5.1, 8);  
double y = Max(8, 5.1);
```

You might be tempted to assume that a template's return type can be deduced from the context in which the template is used. Unfortunately, C++ compilers don't use the context (i.e. the type on the left hand side of the assignment) during template instantiation and therefore the above attempt will fail to compile.

Ambiguous Parameters

- **Compiler deduces parameters**
 - according to language definitions
- **Easy to confuse *const char** and *string***
 - double quotes implies *const char**

```
template <typename T>
T Max(T a, T b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

```
int main()
{
    string s = Max("Red", "Blue");
}
```

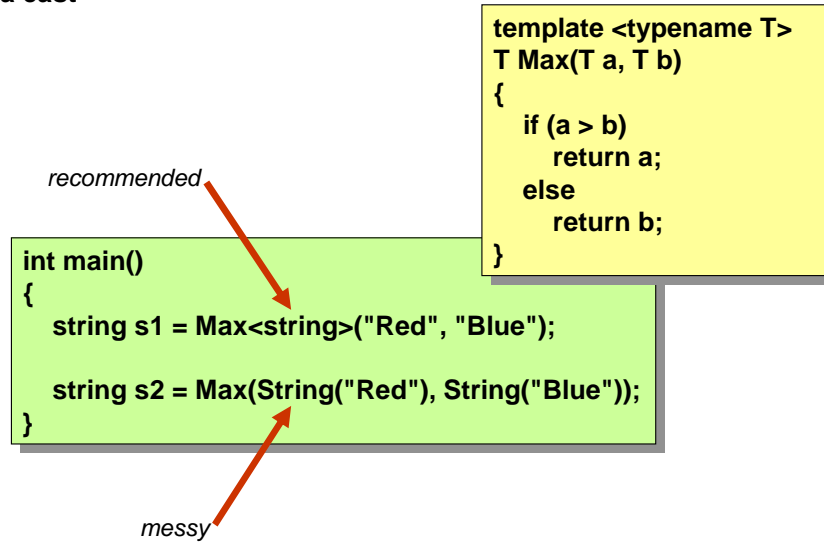
will compare pointers



The compiler will always use language definitions to determine template parameter types. Sometimes this can be a little confusing. For example, if you pass two "strings" to Max using double quotes, the compiler will deduce that the parameters are by definition *char** and not *string*. What is worse is that the Max function will find the larger of the two pointers and not compare the strings. Definitely not what you wanted!

Unambiguous Notation

- Can always specify template parameters in the call
 - or use a cast



Copyright ©1994-2010 CRS Enterprises

54

To avoid ambiguities on parameter type deduction you can state the type explicitly, as in

```
string s1 = Max<string>("Red", "Blue");
```

or if you prefer, use a cast

```
string s2 = Max(String("Red"), String("Blue"));
```

Usually the former notation is clearer and therefore recommended.

Special Cases

- **special cases can be defined**

- compiler always prefers a non template function
- exact parameter matches preferred to casting

```
Max<string>("Red", "Blue");
Max("Red", "Blue");
```

```
template <typename T>
T Max(T a, T b)
{
    if (a > b)
        return a;
    else
        return b;
}

const char* Max(const char* a, const char* b)
{
    if (strcmp(a,b) > 0)
        return a;
    else
        return b;
}
```

Copyright ©1994-2010 CRS Enterprises

55

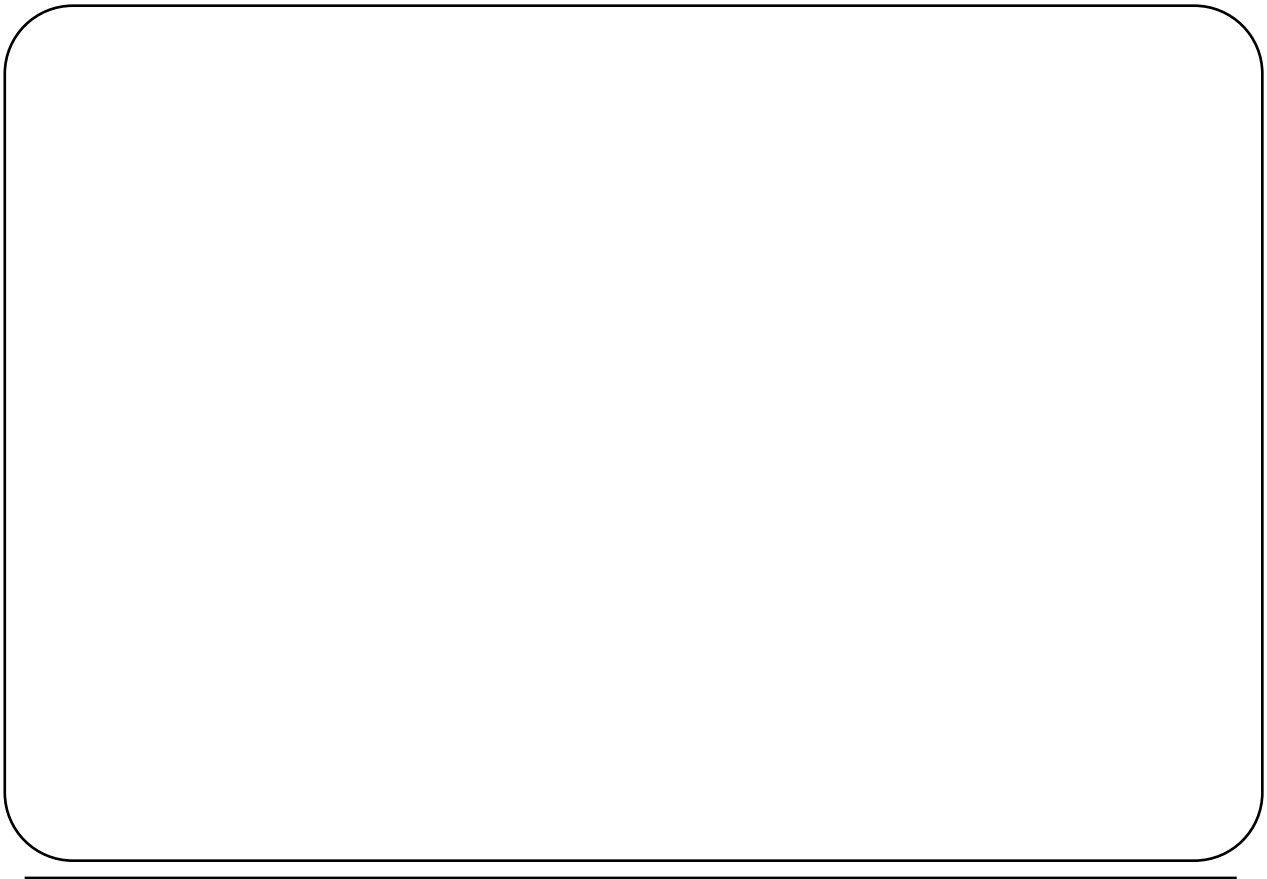
In many situations you would like to use a generic algorithm, but special cases exist.

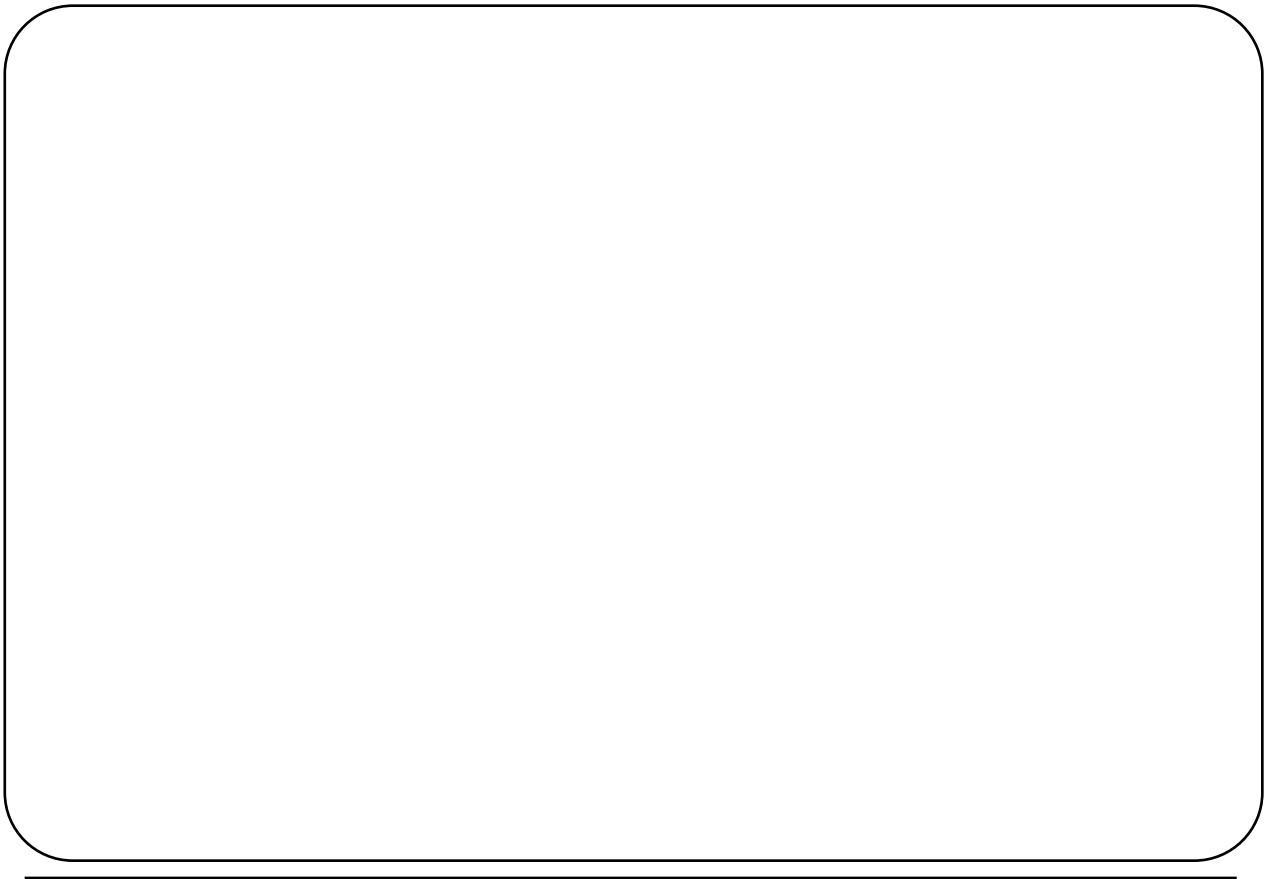
No problem. Define a template for the generic case and provide an explicit function (specialization) for the special case. The compiler will always select non-template functions if each parameter exactly matches the function signature.

In cases where the parameter types don't quite match, the compiler will attempt to cast to resolve the ambiguity. This leads to four cases

- non-template function with exact parameter matches
- template function with exact parameter matches
- non-template function requiring casts to fix parameter matches
- template function requiring casts to fix parameter matches

The compiler will attempt to match function calls by working down the above list.





Chapter 6

6

Template Classes

- **Basics of Template Classes**

- declaration
- instantiation

- **Template Parameters**

- typename
- values
- template parameters
- defaults

- **Specialization**

- partial specialization
- full specialization



Chapter 6

Generic Classes

- **Similar Classes**
 - ready to be converted to templates

```
class ArrayOfInts
{
private:
    int array[3];
public:
    ArrayOfInts(int);
    void Print( );
};

class ArrayOfDoubles
{
private:
    double array[3];
public:
    ArrayOfDoubles(double);
    void Print( );
};

int main( )
{
    ArrayOfInts a(3);
    ArrayOfDoubles b(5.5);
}
```

- **identify types**
 - that can be made generic

Copyright ©1994-2010 CRS Enterprises

60

Consider the two classes `ArrayOfInts` and `ArrayOfDoubles` shown above. These classes have a great deal in common and are suitable candidates for being converted into a class template.

However, with class templates we have more work to accomplish than with function templates. Not only do we need to parameterize the class definitions, but we need to write templates for all common methods, including constructors.

Converting to Templates

- Create Templates for
 - class definition
 - each member function
- Parameters can be classes or built in types
 - use **class** or **typename**

```
template <class T>
class Array
{
private:
    T array[3];
public:
    Array(T);
    void Print( );
};
```

```
template <class T>
Array<T>::Array(T x0)
{
    for (int i = 0; i < 3; i++)
        array[i] = x0;
}

template <class T>
void Array<T>::Print( )
{
    for (int i = 0; i < 3; i++)
        cout << array[i] << 't';
    cout << endl;
}
```

Copyright ©1994-2010 CRS Enterprises

61

Converting the two classes ArrayOfInts and ArrayOfDoubles to templates is straightforward. Note that a template is required not only for the class definition, but also for each member function.

The template parameter does not have to be a class, despite the class keyword. In fact the template parameter can be either a class or a built in type. You can use the typename keyword in place of class; the two are synonymous.

Instantiating Templates

- **Template instantiation**

- compiler instantiates a new class for each call
- linker removes duplicates

```
int main( )
{
    Date newYear(1, 1, 2000);

    Array<int> a(8);
    Array<double> b(8.888);
    Array<Date> c(newYear);

    a.Print( );
    b.Print( );
    c.Print( );
}
```

```
template <typename T>
class Array { ... }

template < typename T>
Array<T>::Array(T x0) { ... }

template < typename T>
void Array<T>::Print( ) { ... }
```

Copyright ©1994-2010 CRS Enterprises

62

All that remains is to instantiate the templates in a calling program. Types are passed inside angle brackets

```
Array<int> a(8);
Array<double> b(8.888);
Array<Date> c(newYear);
```

The compiler instantiates a separate class for each of these calls, along with the associated member functions. In the case where a template is instantiated in several compilation modules this will lead to duplicate class definitions; the linker will remove any unnecessary code. This scheme is used by the majority of compiler/linker systems, other systems handle template instantiation differently from above.

Template Parameters

- 3 Categories of Template Parameters

- Typenames
 - a user defined type (e.g. Date)
 - built in type (e.g. int)
- Values
 - the value of a type (e.g. 3)
- Template
 - templates can accept other templates as parameters
 - beware syntax pitfall with double > >

note the space in > >

```
template <typename T, template <typename U> >  
// class definition that uses T and U
```

Copyright ©1994-2010 CRS Enterprises

63

Templates can take any number of parameters and each parameter can come from one of three categories.

Typename

We have already seen the use of typenames as template parameters.

Value

Parameters can be simple values, the same as in function calls.

Template

Templates can accept other templates as parameters. This can lead to complex and intriguing designs. Much of the power of C++'s template mechanism lies in the use of these so called template template parameters. Be careful to leave a space between the closing > of the two templates; the compiler assumes >> is the right shift operator.

Using Values as Parameters

- **template Value parameters**
 - value must be known at compile time

```
int main( )
{
    Date newYear(1, 1, 2000);

    Array<int, 3> a(8);
    Array<double, 5> b(8.888);
    Array<Date, 4> c(newYear);

    a.Print( );
    b.Print( );
    c.Print( );
}
```

```
template <typename T, int SIZE>
class Array
{
private:
    T array[SIZ];
public:
    Array(T);
    void Print( );
};

template < typename T, int SIZE>
Array<T , SIZE>::Array(T x0)
{
    for (int i = 0; i < SIZE; i++)
        array[i] = x0;
}

template < typename T, int SIZE>
void Array<T , SIZE >::Print( )
{
    for (int i = 0; i < SIZE; i++)
        cout << array[i] << '\t';
    cout << endl;
}
```

Copyright ©1994-2010 CRS Enterprises

64

As mentioned previously, template parameters are not restricted to typenames. In the example above, the size of the array is parameterised in the template

```
template <typename T, int SIZE>
```

The template defines an array of type T with a SIZE specified at compile time. You can see why templates are so popular - with minimal effort we have created an extremely powerful set of classes!

Note that the class name is becoming a little complicated. The constructor

```
template <typename T, int SIZE>
```

```
Array<T , SIZE>::Array(T x0)
```

refers to the class

```
Array<T , SIZE>
```


Default Parameters

- default parameters
 - similar to functions

```
template <typename T = int , int SIZE = 6>
class Array
{ ... };
```

```
int main( )
{
    Array<int, 6> a(1);
    Array<int> b(2);
    Array<> c(3);
    ...
}
```

```
template <typename T = int , int SIZE = 6>
class Array
{
private:
    T array[SIZE];
public:
    Array(T);
    void Print( );
};

template < typename T, int SIZE>
Array<T , SIZE>::Array(T x0)
{
    for (int i = 0; i < SIZE; i++)
        array[i] = x0;
}

template < typename T, int SIZE>
void Array<T , SIZE >::Print( )
{
    for (int i = 0; i < SIZE; i++)
        cout << array[i] << 't';
    cout << endl;
}
```

Copyright ©1994-2010 CRS Enterprises

65

A nice addition to templates is the ability to supply default parameters. This works in much the same way as with functions.

More advanced use of templates often involves defining several template parameters, with many of the parameters providing specialist behavior. It is really convenient if most of the template parameters can be defaulted to sensible values; then the casual user needs only to supply the essential parameters and let the template decide on default values for the rest. The specialist user still can customize the template by supplying every parameter.

Template Specialization

- **Partial Specialization**
 - specialized version of template
 - only called for a subset of types
 - e.g. pointer types
- **Full Specialization**
 - specialized version of template
 - only called for a defined type
- **Both specializations ...**
 - generic template is still required
 - but only needs to be declared
 - not necessary to define if not used

Copyright ©1994-2010 CRS Enterprises

66

Advanced techniques with templates rely on template specialization. You can define a generic template as per the previous slides, but then provide special cases of the template, or specializations, with a completely different code base.

There are two forms of template specialization

partial specialization

With partial specialization, a subset of possible template parameters will use the specialized template, but everything else will use the generic template. For example you can define a template that takes a typename T with a specialization for all pointer types T*.

full specialization

This is similar to partial specialization, except the special case must be uniquely defined. For example you can define a template that takes a typename T with a specialization for the case where T is an int.

It should be noted that the generic template is still required, even if you only want to instantiate the specialized types. If you never intend to instantiate the generic template, you are allowed to merely declare the template and not provide any implementation.

Full Specialization

- **Specialize on type**
 - generic template for **T**
 - specialization for **int**

```
int main()
{
    collection<string> list1;
    collection<int> list2;
    collection<bool> list3;

    list1.f1();
    list2.f2();
    list3.f1();
}
```

```
// the generic template
template <typename T>
class collection
{
public:
    void f1() {}
    void f2() {}
    void f3() {}
};

// specialization for bool
template <>
class collection<bool>
{
public:
    void f1() {}
    void f2() {}
    void f3() {}
};
```

Copyright ©1994-2010 CRS Enterprises

67

This example shows how full specialization of templates works. The generic template is defined, followed by the specialization. The specialization parameter is added to the class definition

```
class collection<bool>
```

Note the template clause has no parameters defined

```
template <>
```

Typically, full specializations will be provided for a number of types. All specializations must follow the declaration of the generic template.

As before, the code base for the generic and full specialization are unrelated. If you only intend to use the fully specialized templates, the generic template must still be declared as

```
// the generic template
```

```
template <typename T> class collection;
```

Partial Specialization

- **Specialize on pointer types**

- generic template for **T**
- specialization for **T***

```
int main()
{
    collection<string> list1;
    collection<double> list2;
    collection<int*> list3;

    list1.f1();
    list2.f2();
    list3.f1();
}
```

```
// the generic template
template <typename T>
class collection
{
public:
    void f1() { ... }
    void f2() { ... }
    void f3() { ... }
};

// partial specialization for pointers
template <typename T>
class collection<T*>
{
public:
    void f1() { ... }
    void f2() { ... }
    void f3() { ... }
};
```

Copyright ©1994-2010 CRS Enterprises

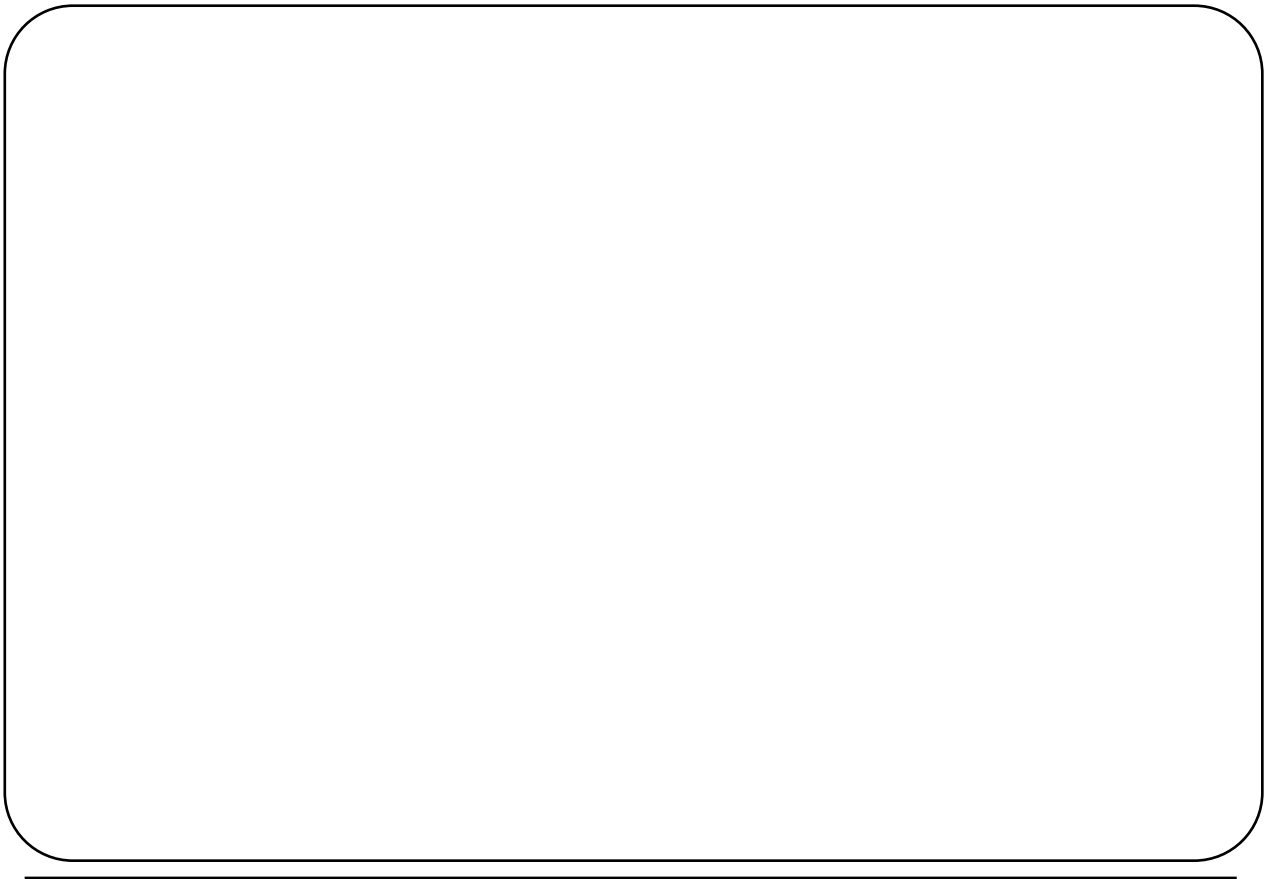
68

This example shows how partial specialization of templates works. The generic template is defined, followed by the specialization. The specialization parameter is added to the class definition

```
class collection<T*>
```

The code base for the generic template and for the partial specializations are unrelated. If you only intend to use the specialization templates, the generic template must still be declared, but not defined, as in

```
// the generic template
template <typename T> class collection;
```



Chapter 7

7

Advanced Templates

- **Template Meta-Programs**
 - compile time algorithms
- **Expression Templates**
 - optimizing complex expressions
 - building expression parse trees at runtime
- **Typelist**
 - manipulation of lists of types
 - template-template parameters



Chapter 7

Part 1 : Template Meta-Programs

- **Program 'runs' at compile time**
 - recursive algorithms
 - use templates
 - generic template defines algorithm
 - template specialization to end the recursion
- **Performance**
 - meta-programs can gain a big performance gain over traditional runtime programs
- **Examples**
 - Factorial
 - Fibonacci
 - Bubble Sort

Copyright ©1994-2010 CRS Enterprises

72

Template meta-programs are specialized programs that are computed during the compilation. These programs make extensive use of templates and recursive algorithms. Compile time programs are analogous to the #define macro expansions familiar from C, and like their C counterpart, these meta-programs can give a significant performance boost to their runtime equivalents.

We will investigate three typical examples

Factorial

Fibonacci

Bubble Sort

Factorial Meta-Program

- **Recursion**

- Generic template for general case
 - $F(N) = N * F(N-1)$
- Full specialization for special case
 - $F(1) = 1$

```
template<int N>
class Factorial {
public:
    enum { value = N * Factorial<N-1>::value };
};

template<>
class Factorial<1>
{
public:
    enum { value = 1 };
};

int main()
{
    int x;
    x = Factorial<6>::value;
}
```

Copyright ©1994-2010 CRS Enterprises

73

All meta-programs are computed at compile time and use the same basic principles. Obviously, ordinary variables cannot be used in a compile time program, so the alternative is to use templates with recursion. It turns out that enums are the only entity available for performing calculations at compile time with the implication that meta-programs are restricted to integer arithmetic.

In the above example the generic template defines an enum value in terms of an enum value from a related template using a recursive relation

$$\text{Factorial}\langle N \rangle::\text{value} = N * \text{Factorial}\langle N-1 \rangle::\text{value}$$

The recursion is broken by a fully specialized template

$$\text{Factorial}\langle 1 \rangle::\text{value} = 1$$

To test the meta-program we can assign the compile time constant

$$\text{Factorial}\langle 6 \rangle::\text{value}$$

to x and either inspect the variable at runtime in a debugger or add a print statement.

Fibonacci Meta-Program

- **Recursion**

- Generic template for general case
 - $F(N) = F(N-1) + F(N-2)$
- Full specialization for special case
 - $F(2) = 1$
 - $F(1) = 1$

```
int main()
{
    int x[4];
    x[0] = Fibonacci<1>::value;
    x[1] = Fibonacci<2>::value;
    x[2] = Fibonacci<3>::value;
    x[3] = Fibonacci<4>::value;
}
```

```
template<int N>
class Fibonacci {
public:
    enum { value = Fibonacci<N-1>::value + Fibonacci<N-2>::value };
};
```

```
template<>
class Fibonacci<1>
{
public:
    enum { value = 1 };
};
```

```
template<>
class Fibonacci<2>
{
public:
    enum { value = 1 };
};
```

Copyright ©1994-2010 CRS Enterprises

74

This is a slightly more complicated meta-program. This time the generic algorithm is defined as

$$\text{Fibonacci}\langle N \rangle::\text{value} = \text{Fibonacci}\langle N-1 \rangle::\text{value} + \text{Fibonacci}\langle N-2 \rangle::\text{value}$$

The recursion is broken by two fully specialized templates

$$\text{Fibonacci}\langle 1 \rangle::\text{value} = 1$$

$$\text{Fibonacci}\langle 2 \rangle::\text{value} = 1$$

Bubble Sort Meta-Program

- **IntBubbleSort**

- calls itself and
- call **IntBubbleSortLoop** template

```
template<int N>
struct IntBubbleSort
{
    static inline void sort(int* data)
    {
        IntBubbleSortLoop<N-1,0>::loop(data);
        IntBubbleSort<N-1>::sort(data);
    }
};

template<>
struct IntBubbleSort<1>
{
    static inline void sort(int* data) { }
};
```

```
int main()
{
    int a[] = { 21, 2, 33, 4, 25 };
    IntBubbleSort<5>::sort(a);
}
```

Copyright ©1994-2010 CRS Enterprises

75

To see what can be achieved by a meta-program we will investigate Bubble Sort. This time the generic template

```
template<int N> struct IntBubbleSort
```

is defined in terms of two other templates

```
template<int I, int J> class IntBubbleSortLoop
```

```
template<int I, int J> struct IntSwap
```

The generic template defines a sort method which is dependent on these sub templates.

As always, the recursion breaker is a specialization of the generic template

```
template<> struct IntBubbleSort<1>
```

which defines an empty sort method.

BubbleSortLoop Template

```
template<int I, int J>
class IntBubbleSortLoop
{
private:
    enum { go = (J <= I-2) };
public:
    static inline void loop(int* data)
    {
        IntSwap<J,J+1>::compareAndSwap(data);
        IntBubbleSortLoop<go ? I : 0,
            go ? (J+1) : 0>::loop(data);
    }
};

template<>
struct IntBubbleSortLoop<0,0>
{
    static inline void loop(int*) { }
};
```

- calls **IntSwap**

simulates an if statement

Copyright ©1994-2010 CRS Enterprises

76

The sub template

template<int I, int J> class IntBubbleSortLoop
defines a loop method which calls the template

template<int I, int J> struct IntSwap
and makes a recursive call to itself.

An interesting feature of this implementation is how the specialized template

template<0,0> class IntBubbleSortLoop
gets called. Notice enum go is defined as a boolean dependent on the expression

$J \leq I-2$

and the recursive call uses the enum go to decide whether to call

IntBubbleSortLoop<I,J+1>::loop(data)

or

IntBubbleSortLoop<0,0>::loop(data)

IntSwap Template

- **Last template in chain**

- calls non template function swap

```
template<int I, int J>
struct IntSwap
{
    static inline void compareAndSwap(int* data)
    {
        if (data[I] > data[J])
            swap(data[I], data[J]);
    }
};

inline void swap(int& a, int& b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

Copyright ©1994-2010 CRS Enterprises

77

The remainder of the meta-program is shown above. This final template calls an ordinary method swap.

It is an interesting exercise to work through the template expansion for small values of I. You will find

IntBubbleSort<3>::sort(a);

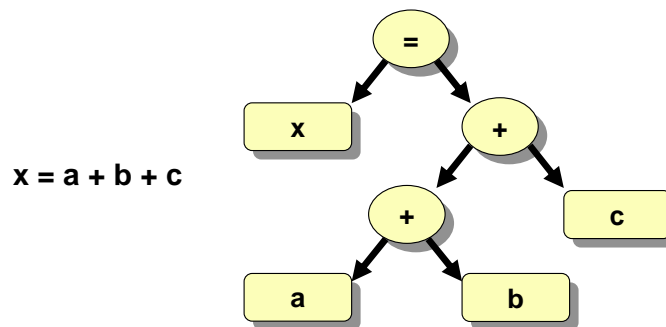
translates to 3 inline calls to sort and

IntBubbleSort<4>::sort(a);

translates to 6 inline calls to sort.

Part 2 : Expression Templates

- **High Performance C++ Applications**
 - must avoid excessive use of temporary objects
 - compiler doesn't optimize operator overloaded expressions
- **Expression Templates**
 - allow class to build an expression parse tree at run time
 - can optimize away unnecessary temporary objects



Copyright ©1994-2010 CRS Enterprises

78

Over the years, many C++ class libraries have been developed for linear algebra (Matrix) applications and while these class libraries provide the necessary functionality they all suffer from computational inefficiencies. These inefficiencies stem from the way C++ performs expression evaluation, namely the introduction of temporaries at each stage of an expression evaluation. With matrix calculations this can mean creating large intermediate matrices which are discarded at the end of the calculation.

The blitz++ library was the first library to recognise that these inefficiencies could be eliminated by building an expression parse tree at run time and delaying evaluation of an expression until the parse tree is complete.

The Problem

- **temporary array created**
 - as part of expression evaluation
 - unnecessary additions
 - for **every** element in the two dimensional array
- **evaluation proceeds (for each element) ...**

temp = a + b

x = temp + c

```
int main()
{
    Array a, b, c, x;
    x = a + b + c;
}
```

```
Array operator+(Array const& a, Array const& b)
{
    std::size_t const n = a.size();
    Array result;

    for (std::size_t row = 0; row != n; ++row)
        for (std::size_t col = 0; col != n; ++col)
            result[row][col] = a[row][col] + b[row][col];

    return result;
}
```

Copyright ©1994-2010 CRS Enterprises

79

Consider the above example where we are adding together 3 matrices. Using traditional techniques, each element in the resulting array x, will be computed using a temporary

temp = a + b

x = temp + c

and this will be true for each row and column of the matrix.

However, using expression templates we can reduce the calculation to the optimal

x = a + b + c

Expression Templates

- **expression templates**
 - used to build parse tree
- **objects of type *Expression***
 - added to parse tree

```
struct plus;
struct minus;
```

```
template <typename L, typename R>
Expression<L,plus,R> operator+(const L& l,const R& r)
{
    return Expression<L,plus,R>(l,r);
}
```

 $l + r$

```
template <typename L, typename R>
Expression<L,minus,R> operator-(const L& l,const R& r)
{
    return Expression<L,minus,R>(l,r);
}
```

 $l - r$

Expression templates are used to build a parse tree at runtime. The expression templates generate objects that encapsulate a left operand (l), an operator (plus and minus) and a right operand (r). Note that these objects capture the intention of the expression, but do not perform any calculation.

Example

- **One dimensional array expressions**
 - does not use temporaries
 - build parse tree of expression
 $a + b + d - c$
 - evaluate parse tree at point of assignment
- **Array**
 - is wrapper for an array of double

```
int main()
{
    Array result, a, b, c, d;
    result = a + b + d - c;
}
```

```
struct Array
{
private:
    std::vector<double> array;
};
```

Copyright ©1994-2010 CRS Enterprises

81

Let's investigate expression templates for use with one dimensional array calculations

struct Array

Our object is to perform calculations that avoid the use of temporaries by building a parse tree for the expression

$a + b + d - c$

More Expression Templates

- **overloading []**
 - needed to extract array elements
 - trigger calls to *apply*

```

template <typename L, typename Op, typename R>
struct Expression
{
    Expression(const L& l, const R& r)
        : l(l), r(r) {}

    double operator[](unsigned index) const
    {
        return Op::apply(l[index], r[index]);
    }

    const L& l;
    const R& r;
};

struct plus
{
    static double apply(double a, double b)
    {
        return a + b;
    }
};

struct minus
{
    static double apply(double a, double b)
    {
        return a - b;
    }
};
  
```

The diagram illustrates how an `Expression` template uses an `Op` (operator) template parameter to call `Op::apply` within its `operator[]` method. Two concrete operators, `plus` and `minus`, are shown, each implementing the `apply` static method to perform addition and subtraction respectively. Red dashed arrows indicate the call from `Op::apply` in the `Expression` template to the corresponding `plus::apply` or `minus::apply` methods.

Copyright ©1994-2010 CRS Enterprises

82

Ultimately, expression templates must call methods that perform the actual calculation. In our case this is achieved through the overloaded `[]` operator. When the runtime expression parse tree is evaluated (see subsequent slides), the `apply` method will be called with two operands. The match on the `Op` template parameter will direct calls to either `plus::apply` or `minus::apply`.

Triggering the Evaluation

- **operator=()**
 - triggers the evaluation
 - parse tree already built
 - no temporaries in calculation

```
struct Array
{
private:
    std::vector<double> array;
public:
    template <typename Expr>
    Array& operator=(const Expr& rhs)
    {
        for(unsigned i = 0; i < array.size(); i++)
            (*this)[i] = rhs[i];
        return *this;
    }
};
```

Copyright ©1994-2010 CRS Enterprises

83

With expression templates, the parse tree gets built as the plus and minus operators are encountered. Recall that no computations are made during this phase.

When the right hand side of an expression has been fully parsed, the runtime parse tree will be complete. It is at this point that the assignment operator is parsed; this will call the overloaded method shown above; this in turn calls the overloaded [] operator and the parse tree is unwound (as shown previously) and the calculation proceeds.

In summary, the parse tree is built as the right hand side of the expression is parsed by our templates and evaluation takes place only when the assignment operator is encountered.

Part 3 : Traits

- ??

The Problem

- Return value not considered deterministic
 - compiler can't deduce RET

```
template <typename L, typename R, typename RET>  
RET Max(L a, R b)  
{  
    if (a > b)  
        return a;  
    else  
        return b;  
}
```

```
double x = Max(5.1, 8);  
double y = Max(8, 5.1);  
float f1 = Max(5.1F, 8);  
float f2 = Max(8, 5.1F
```

Using Specialization

- Define each special case

- Max(int, float) returns float
- Max(int, double) returns double
- Max(float, int) returns float
- Max(double, int) returns double

define as a specialized template

```
template <typename L, typename R>  
typename promote_trait<L,R>::T_promote Max(L a, R b)  
{  
    if (a > b)  
        return a;  
    else  
        return b;  
}
```

The promote_trait Template

- **Generic case ..**

- must be defined,
- but not used

```
// generic case
template<class A, class B>
class promote_trait {
public:
    typedef void T_promote;
};
```

- **Full Specialization**

- for each case
- does the real work

```
template<>
class promote_trait<int, double> {
public:
    typedef double T_promote;
};
```

```
template<>
class promote_trait<double, int> {
public:
    typedef double T_promote;
};
```

Part 4 : Typelists

- **Compile time list of types**
 - defined in terms of **typelist** templates
 - uses template-template parameters
- **Write Meta-programs**
 - to manipulate typelists
 - is given type in list?
 - how many types are in the list?
 - append type to list
- **Can be used in Advanced Applications**
 - just an introduction in these pages
 - for more details consult
 - **Modern C++ Design - Andrei Alexandrescu**
Addison-Wesley, ISBN 0201704315

Copyright ©1994-2010 CRS Enterprises

88

Typelists are an advanced template technique for manipulating lists of types at compile time. Typelist meta-programs can be written to perform all sorts of operations on these lists, such as

checking if a given type is in the list

calculating how many types are in the list

appending a new type to the list

Typelists are in themselves just a building block. They can be used to good effect in advanced applications that deal with lists of arbitrary types. Full details can be found in the excellent text

Modern C++ Design

Andrei Alexandrescu

Addison-Wesley, ISBN 0201704315

Definition of a Typelist

- **template *Typelist***
 - has a head, *H*
 - has tail, *T*
- **Recursion**
 - longer lists built using recursion
 - use Typelist as parameter to another Typelist
- **NullType**
 - no content
 - just used to indicate end of list

```
class NullType {};
```

```
template <typename H, typename T>
struct Typelist
{
private:
    typedef H Head;
    typedef T Tail;
};
```

```
Typelist<int, <Typelist<X, Typelist<double, NullType> > >
```

Copyright ©1994-2010 CRS Enterprises

89

Typelists are defined in terms of the template

```
template <typename H, typename T> struct Typelist
```

This template takes two parameters, a head, *H* and tail, *T*. The head is invariably another typelist (a typelist is a typename) and the tail is a single type (NullType) that terminates the list. The NullType class has no content and is only ever used as a marker at the end of a list.

Given that a typelist will normally consist of several types and that the Typelist template only has two parameters, we must use recursion to generate longer lists. That this can be done, relies on C++'s ability to use a template as a parameter to another template. In our case this is done recursively, the Typelist is used as the head of another Typelist template.

Example Typelists

- **use #defines**
 - to simplify notation

```
#define TYPELIST_1(T1) Typelist<T1, NullType>
#define TYPELIST_2(T1,T2) Typelist<T1, TYPELIST_1(T2)>
#define TYPELIST_3(T1,T2,T3) Typelist<T1, TYPELIST_2(T2,T3)>
#define TYPELIST_4(T1,T2,T3,T4) Typelist<T1, TYPELIST_3(T2,T3,T4)>
```

```
typedef TYPELIST_1(int) MyTypeList1;
typedef TYPELIST_2(int, double) MyTypeList2;
typedef TYPELIST_3(int, double, string) MyTypeList3;
typedef TYPELIST_4(int, double, string, float) MyTypeList4;
```

Using templates as parameters to other templates is essential for typelists. Unfortunately, a long list will have many nested brackets (< >) to represent the various templates. While this is syntactically valid, it leads to unwieldy code and #defines can be used to simplify the notation considerably.

Length of a Typelist

- **Generic template**
 - required, but never used
- **Non empty lists**
 - specialize on HEAD = Typelist
 $L = 1 + L(\text{TAIL})$
 - specialize on empty list
 $L(\text{NullType}) = 0$

```
// generic declaration (never used)
template <typename TList>
struct Length;

template <typename H, typename T>
struct Length<Typelist<H,T> >
{
    enum {value = 1 + Length<T>::value};
};

template<>
struct Length<NullType>
{
    enum { value = 0 };
};
```

```
typedef TYPELIST_4(int, double, string, float) MyTypeList4;
int n = Length<MyTypeList4>::value;
```

Copyright ©1994-2010 CRS Enterprises

91

Typelist meta-programs all follow similar designs; the above example of determining how many types are in a typelist is typical. Recall that typelists are always defined recursively in terms of other typelists. However, the Typelist template is defined for all types including types that are not typelists (i.e. no NullType). To overcome this problem we employ a crafty trick. We declare the generic template but do not define it, and instead provide specializations for cases where the head of the typelist is itself a typelist. This means only the specialization templates can be used in our code, which of course is exactly what we want.

As for the recursion, we define the length of a typelist as one more than the length of its head (also a typelist). Eventually, the typelist reduces to the special case of just a NullType (regarded as the degenerate typelist).

IndexOf type in a Typelist

- **Generic template**
 - required, but never used
- **Specializations**

```
Index(NullType) = -1
Index(T, NullType) = 0
Index(T) = Index(T-1) + 1
```

```
// generic declaration (never used)
template<typename TList, typename T>
struct IndexOf;
```

```
template<typename T>
struct IndexOf<NullType,T>
{ enum { value = -1 }; };
```

```
template<typename T, typename Tail>
struct IndexOf<Typelist<T,Tail>,T>
{ enum { value = 0 }; };
```

```
template<typename Head, typename Tail, typename T>
struct IndexOf<Typelist<Head,Tail>,T>
{
    enum { temp = IndexOf<Tail,T>::value };
    enum { value = temp == -1 ? temp : temp + 1 };
};
```

```
typedef TYPELIST_4(int, double, string, float) MyTypeList4;
n = IndexOf<MyTypeList4,string>::value;
```

Copyright ©1994-2010 CRS Enterprises

92

Determining the position of a type in a typelist proceeds along similar lines. The generic Typelist template is declared, but not defined, and specializations are provided for typelists only. This time the code is slightly more involved because we need to consider the case where the type is not in the list (enum value = -1). Note we employ our old trick with enum temp to simulate an if statement during the recursion.

Part 5 : Template Template Parameters

Template Parameters

- **Template parameters must be:**
 - **1. Types**
 - `template t1<T1, T2, T3, T4>`
 - where T1=int, T2=double, T3=Money, T4=Date
 - **2. Values**
 - `template t2<int N, double D>`
 - where N=3, D=5.87 (compile time constant)
 - **3. Other templates**
 - `template t3< template t1<....> >`
 - beware closing > >
 - extensive use of recursion

Template Template Parameters

- Start with some ordinary templates

```
template <class T> struct _4ple
{
    T t1;
    T t2;
    T t3;
    T t4;
};
```

```
template <class T> struct _3ple
{
    T t1;
    T t2;
    T t3;
};
```

```
template <class T> struct _2ple
{
    T t1;
    T t2;
};
```

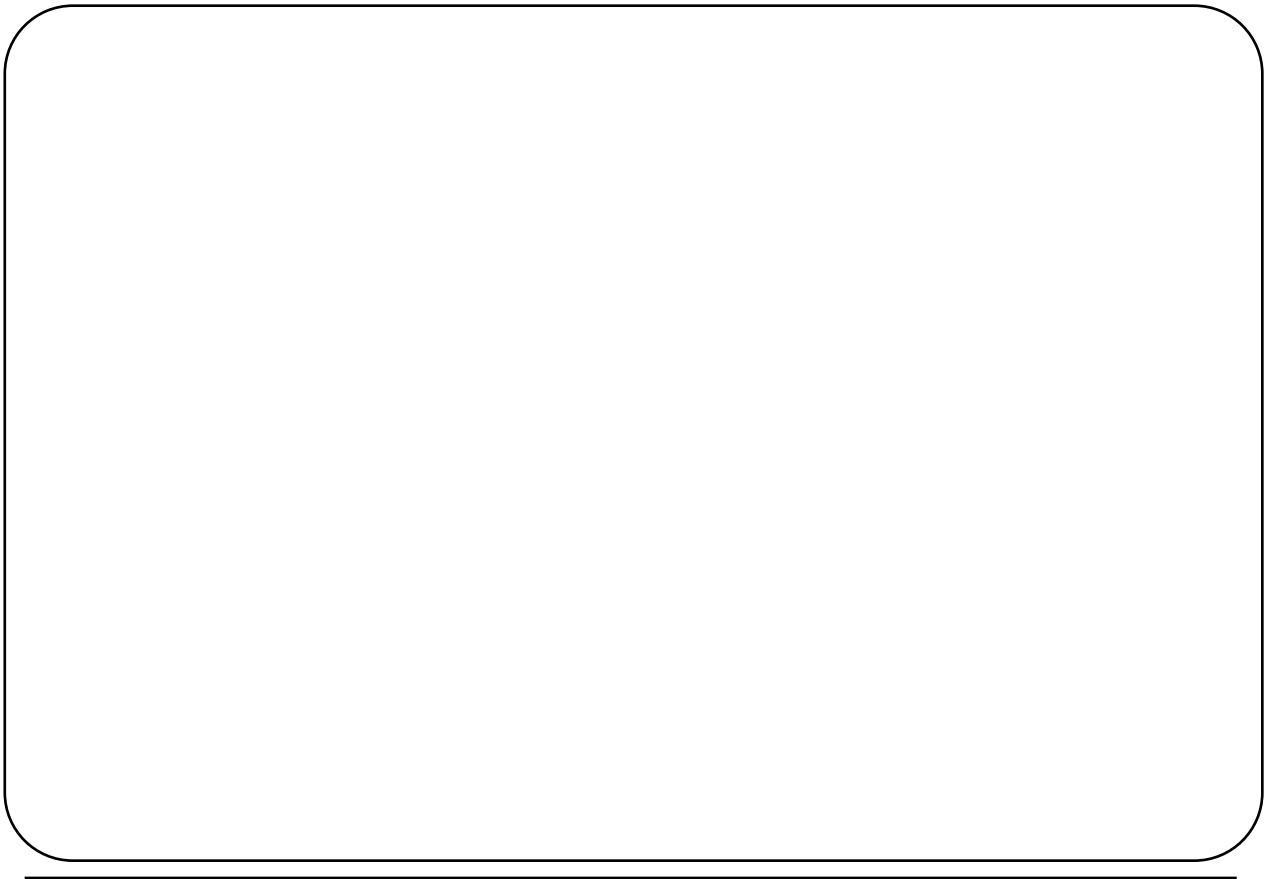
Define Template of Templates

- Use these templates as parameters to other templates

dummy name is never used

```
template
<template<typename dummy> class T, typename S>
struct builder
{
    T<S> u1;
    T<S> u2;
};
```

```
int main()
{
    builder<_2ple,double> m2;
    builder<_3ple,int> m3;
    builder<_4ple,string> m4;
    ...
}
```

Chapter 8

8

Handle-Body

- **The idiom**
 - separation of interface and implementation
- **Reference Counting**
 - string class example
- **Retro-fitting Reference Counting**
 - to arbitrary class
- **Dynamic association**
 - changing roles

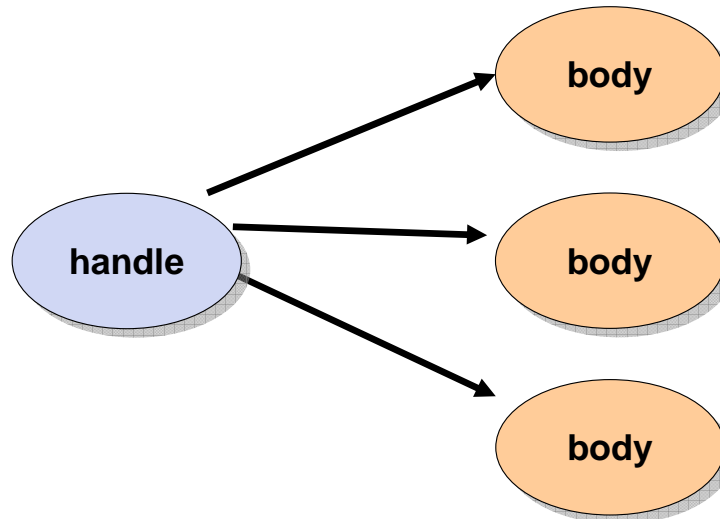


Chapter 8

Handle-Body Idiom

- **Handle-Body**

- one of the most significant programming idioms in C++
- separate interface (handle) from implementation (body)



Copyright ©1994-2010 CRS Enterprises

100

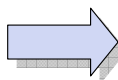
The handle-body design pattern, also known as the bridge, allows the implementation (or body) of an object to be separated from the interface (or handle) through which the object is accessed. The two parts work together to provide one logical abstraction.

There are many design uses of handle-body. For example, since access to the object is through its handle, and given that this is separated from the body, it is possible to change the body at runtime and hence the behaviour of the underlying object. This amounts to a form of dynamic typing. Other examples include reference counted techniques and smart pointers. Many other designs are possible.

Reference Counted Strings

- **Handle**
 - provides public interface for clients
- **Body**
 - provides the reference counted implementation
 - more efficient for sharing strings

```
class string
{
public:
    string(const char* = "");
    string(const string&);
    ~string();
    string& operator=(const string&);
private:
    class body;
    body* self; // cheshire cat
};
```



```
class string::body
{
private:
    char* text;
    size_t count;
public:
    body(const char*);
    ~body();
    void Increment();
    void Decrement();
};
```

Copyright ©1994-2010 CRS Enterprises

101

One of the principal uses of Handle-Body is reference counting. Suppose we wish to design a reference counted string class; this is our own string class and not the string class from the standard library.

Begin by defining a handle that defines the public interface for the string class. Note that the handle defines a pointer, `self`, to the private body; it is convenient to use a nested class for the body. This design is sometimes called the Cheshire Cat, since the class user knows there is an underlying implementation, but all that is visible is the "smile".

Next we define the implementation in the body. The implementation will use reference counting.

Reference Counting: Constructors

- **Constructor for new body**
 - when string is first created
 - or modified
- **Copy Constructor**
 - when we are duplicating string
 - reuse existing body

```
class string
{
public:
    string(const char* = "");
    string(const string&);
    ~string();
    string& operator=(const string&);
private:
    class body;
    body* self; // cheshire cat
};
```

```
string::string(const char* s)
{
    self = new body(s);
}

string::string(const string& s)
{
    self = s.self;
    self->Increment();
}
```

Copyright ©1994-2010 CRS Enterprises

102

The handle class must distinguish between two cases: when a new body class is required and when an existing body class can be reused. The handle should define two constructors, one for each case.

The first constructor is used when a string is created or gets modified; we need to create a new body in this case. Whereas, the copy constructor is used when an object gets copied and the intention is to reuse the existing body.

Adjusting the Reference Count

- When reference count reaches zero
 - body must commit suicide
 - **delete this**

```
string::~~string()
{
    self->Decrement();
}
```

```
void string::body::Increment()
{
    count++;
}

void string::body::Decrement()
{
    count--;
    if (count == 0) delete this;
}
```

```
class string::body
{
private:
    char* text;
    size_t count;
public:
    body(const char*);
    ~body();
    void Increment();
    void Decrement();
};
```

body commits suicide

Copyright ©1994-2010 CRS Enterprises

103

Adjusting the reference count is easy - write Increment and Decrement methods.

The only interesting point is what to do when the reference count on the body reaches zero. Obviously, in this case the body is no longer required and it needs to be deleted. Since the Decrement method is part of the body class, the body must delete itself

delete this

Reference Counting: Assignment

- **Assignment**
 - decrement count on old body
 - increment count on new body
 - check for self assignment
 - return `*this`
 - for cascaded assignments

```
class string
{
public:
    string(const char* = "");
    string(const string&);
    ~string();
    string& operator=(const string&);
private:
    class body;
    body* self; // cheshire cat
};
```

```
string& string::operator=(const string& s)
{
    if (this == &s) return *this;
    self->Decrement(); // old string
    self = s.self;
    self->Increment(); // new string
    return *this;
}
```

check for
self assignment

cascaded
assignments

Copyright ©1994-2010 CRS Enterprises

104

Assignment must be carefully designed. The handle must arrange to decrement the reference count on the old body and increment the reference count on the new body, but the handle must first check for self assignment. If this is not done then there is a danger of the body deleting itself when the reference count is decremented and the increment will fail.

Cascading assignment requires the handle to return a reference to the left operand, as in

```
s1 = s2 = s3
```

which equates to

```
s1.operator=(s2.operator=(s3))
```


Using the Handle

- **Assignment**

- decrement count on old body
- increment count on new body
- check for self assignment
- return *this
 - for cascaded assignments

```
class string
{
public:
    string(const char* = "");
    string(const string&);
    ~string();
    string& operator=(const string&);
private:
    class body;
    body* self; // cheshire cat
};
```

```
int main()
{
    string s1("Blue");
    string t1("Red");

    string s2(s1);
    string s3(s2);
    string t2(t1);

    t2 = t1 = s1;
}
```

char* constructor

copy constructor

Copyright © 1994-2010 CRS Enterprises Ltd

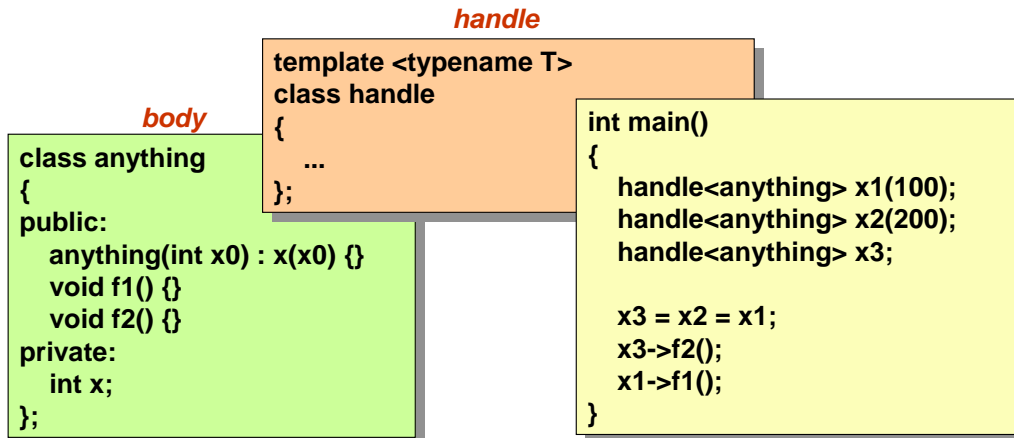
105

An example of using the string class is shown above. The client interfaces directly with the handle class and is unaware of the underlying body.

After s1 and t1 have been created, there will be two separate body objects, each with a reference count of 1. By the time the three copy statements have completed, the string "Blue" will have a reference count of 3 and "Red" a count of 2. By the end of the assignments, "Blue" will have a reference count of 3, but the body for red will have committed suicide (the reference count reached 0).

Generalizing Handle-Body

- Reference Counting can be retro-fitted to any class
 - without modifying the class
 - using templates
 - reusing the basic algorithms of the string class



Copyright ©1994-2010 CRS Enterprises

106

The Handle-Body idiom can be combined with templates to retro-fit reference counting to arbitrary classes without modifying the original class - the original class becomes the body. The basic algorithms remain the same, but some adjustments are required to the handle to avoid necessitating changing the original class.

Template for the Handle

```
template <typename T>
class handle
{
public:
    handle()
    {
        pBody = 0;
        pCount = new int(1);
    }

    template <typename B>
    handle(B theBody)
    {
        pBody = new T(theBody);
        pCount = new int(1);
    }

    handle(const handle<T>& h)
    {
        pBody = h.pBody;
        pCount = h.pObject;
        increment();
    }
    ...
};
```

reference count lives on heap
- not part of handle or body

note extra template parameter
for typename of body

Copyright ©1994-2010 CRS Enterprises

107

The implementation of the template handle class is similar to before. Note that the reference count can't be part of the handle (there are many handles) and can't be part of the body (the body is already written), so it has to be placed on the heap.

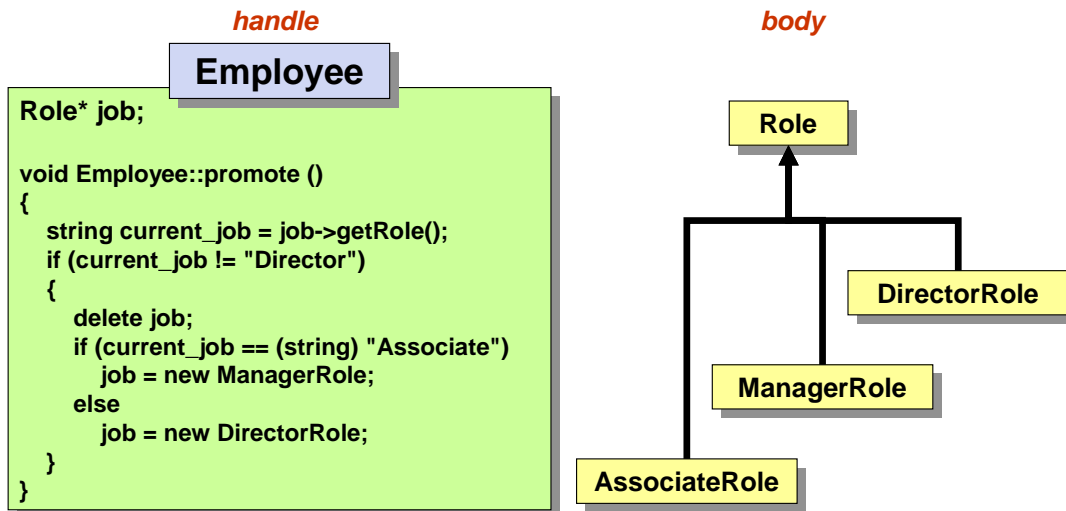
The constructor that creates the body object

```
template <typename B>
handle(B theBody)
```

takes any body class as a parameter.

Dynamic Association

- change Role (*body*) at runtime



Copyright ©1994-2010 CRS Enterprises

108

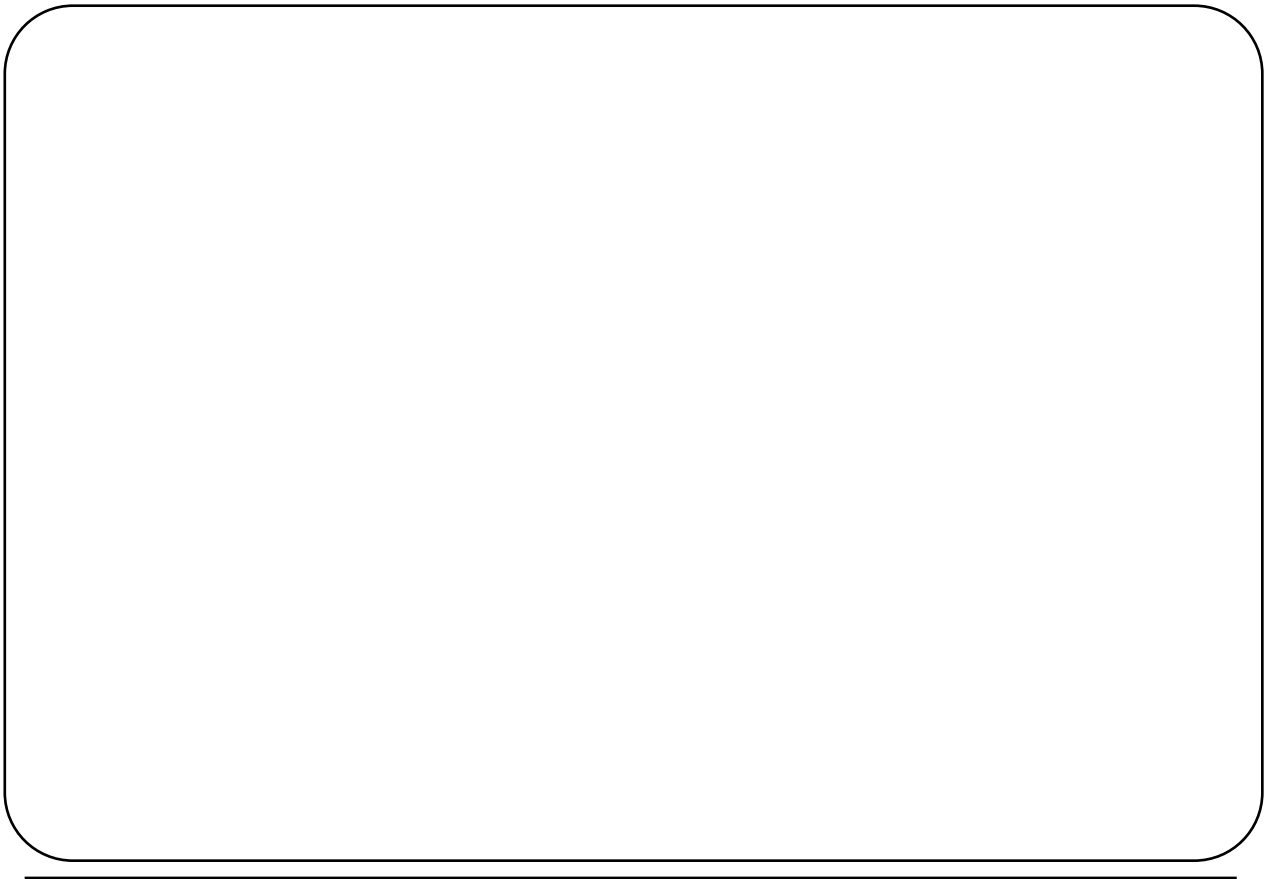
The Handle-Body idiom is not restricted to reference counted designed. As a further example of this important idiom consider the Employee class above.

Suppose we wish our design to have three different types of employee: associate, manager and director. An obvious implementation would be to provide an inheritance hierarchy based on the Employee class, but because of C++'s static nature this design would not allow employees to change their class once defined. What we need is a form of dynamic inheritance.

In practice this can be achieved using dynamic association. The Employee class (handle) provides a private instance variable

Role* job

to associate with the Role class hierarchy (body). By using the job pointer, we can change the Role association at any time, thus achieving the dynamic association we are seeking. The dynamic association is embodied in the promote method.



Chapter 9

9

Managed Pointers

- **Problems with Raw Pointers**
 - memory access violations etc
- **Overloading ->**
 - unary and binary
 - handle-body idiom
- **Smart Pointers**
 - using templates
 - exception aware smart pointers
- **Standard Library auto_ptr**
 - ownership issues
 - copying and assignment
 - const auto_ptr



Copyright ©1994-2010 CRS Enterprises

111

Chapter 9

Raw Pointers

- **C++ pointer types are notorious**
 - null pointer assignments
 - memory leaks
 - access violations
- **Not Object Oriented**
 - pointers are dumb
 - can't recover from exceptions
 - too primitive to participate in some patterns and idoms
- **Managed Pointers**
 - make C++ pointers intelligent
 - using operator overloading of ->
 - many uses ...

Copyright ©1994-2010 CRS Enterprises

112

It is well known that raw C++ pointer types are a notorious source of programming errors. Typical problems include null pointer assignments, memory leaks, access violations and many, many more. Raw pointers can be extremely useful in the right place, but often they are too primitive for the job in hand. All too frequently, raw pointers are used in a non object oriented manner leading to poorly designed code and they can be dangerous to use in the presence of exceptions. The list goes on and on.

What is required is a way to make C++ pointers intelligent. Fortunately, because of the extensive support for operator overloading in the language, this is a feasible task. The idea is to build a class that wraps the raw pointer and provides overloads for all the common pointer operations adding intelligence as appropriate.

The main operator to overload is

->

but assignment, initialization and pointer dereferencing also need to be considered.

Overloading the -> Operator

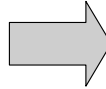
- **Handle-Body idiom**

- -> has special treatment
- overload works in 2 phases

```
Handle h(1, 15);
h->Print();
```

- **Phase 1: unary operator**

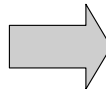
- operates on the *handle*
- returns a C style pointer *pBody*
- for use in next phase to access *body*



```
pBody = handle.operator->()
```

- **Phase 2: binary operator**

- code generated by the compiler
- the C style pointer delegates to the Print() method in the *body*



```
pBody->Print()
```

Copyright ©1994-2010 CRS Enterprises

113

Overloading the -> operator is an interesting operation in its own right and is essential for understanding the mechanics of managed pointers.

The overload of -> is based on the handle-body idiom and works in two phases. In the first phase the operator acts solely on the handle object as a unary operator. Although an operator can be programmed to do anything you like, normal practice is to temporarily extract the wrapped raw pointer from the handle and make this the return value

```
pBody = handle.operator->().
```

In the second phase, the return value is used to delegate to the body, as in

```
pBody->Print()
```

and here the operator is being used as a binary operator. You don't write any code for the second phase; it is generated automatically by the compiler.

Exception Aware Smart Pointers

```
int main()
{
    Handle h1(1, 15);
    Handle h2(2, 30);
    Handle h3(3, 45);
    Handle q;

    h1->Print();
    h2->Print();
    h3->Print();

    try
    {
        q->Print();
    }
    catch(const char* message)
    {
        cout << message << endl;
    }
}
```

- **What if handle has no body?**
 - handle throws an exception

```
class Body
{
    void Print() const;
};

class Handle
{
    const Body* operator->()
    {
        if (pBody == 0) throw "pointer is null";
        return pBody;
    }
    Body* pBody;
};
```

Copyright ©1994-2010 CRS Enterprises

114

A typical example of overloading the `->` operator is to make the managed pointer exception aware. Normally, if you attempt to use a null raw pointer you will get an access violation and a prematurely terminated application.

Using the handle-body idiom allows you to trap the null pointer reference and provide some form of error correction. You could simply throw an exception as shown above or provide some more sophisticated recovery procedure - it's up to you.

Using Templates

- **Handle becomes a template**
 - template parameter T is the body
- **What about copying handles and assignment of handles**
 - copy constructor
 - operator=()
- **Ownership?**
 - body may be shared between several handles
 - which handle owns the body?
- **Is it safe to return raw pointer?**
 - thereby exposing the body

```
template <typename T>
class Handle
{
public:
    Handle() : pBody(new T())
    {}

    ~Handle()
    {
        delete pBody;
    }

    const T* operator->()
    {
        return pBody;
    }

private:
    T* pBody;
};
```

Copyright ©1994-2010 CRS Enterprises

115

Frequently you will want to apply managed pointer techniques to a wide variety of pointer types, but essentially you are writing similar code over and over again. Generic code implies the use of templates and indeed most managed pointer implementations are written as templates.

Templates for managed pointers are not too difficult to write, but you do need to consider copying and assignment and the important issue of ownership. If several handles all point to the same body, then who is responsible for deleting the body? Should you use reference counting, some ownership passing token or what? Each strategy needs careful consideration. Another issue that can cause problems is whether to allow the client access to the underlying raw pointer and hence access to the body. This can be a very dangerous thing to do, but in some strategies it is safe and acceptable. As we will see shortly, the Standard Library provides a managed pointer, `auto_ptr`, that does allow a client access to the body in certain (safe) scenarios.

Auto Cleanup

- **Body lives on the heap**
 - managed by a stack based handle
- **Handle goes out of scope**
 - Handle DTOR
 - deletes body
- **Stack based handles**
 - resource acquisition is initialization

```
template <typename T>
class Handle
{
public:
    Handle() : pBody(new T()) {}
}
```

*body DTOR
called for h2*

*body DTOR
called for h1*

```
void f2()
{
    Handle<Body> h2;
}

void f1()
{
    Handle<Body> h1;
    f2();
}

int main()
{
    f1();
}
```

Copyright ©1994-2010 CRS Enterprises

116

A very useful application of managed pointers is to provide automatic cleanup of heap based objects. Stack based objects have their destructors called by the compiler when they go out of scope, but this doesn't apply to heap based objects. However, by providing a stack based handle, we can arrange to delegate a call to the body's destructor from the handle's destructor. This makes sense, because handles are usually lightweight objects and the overhead of using the stack is acceptable. If we need to cope with many heap based bodies, we can form a stack based collection of handles and provide cascading destructor calls.

Standard Library `auto_ptr`

- **Standard Library `auto_ptr` (deprecated in C++ 2011)**
 - manages body by taking ownership
 - body belongs to only one `auto_ptr`
 - can be used for automatic cleanup of heap based objects

```
void f2()
{
    auto_ptr<person> peter(new person("Peter"));
    f3();
}

void f1()
{
    auto_ptr<person> mary(new person("Mary"));
    f2();
}

int main()
{
    auto_ptr<person> john(new person("John"));
    f1();
}
```

Copyright ©1994-2010 CRS Enterprises

117

The Standard Library provides a template class call `auto_ptr`, expressly for managing the ownership of body objects. Many other strategies are possible, but the central idea behind the `auto_ptr` is that a body can be owned by one and only one `auto_ptr`. This leads to some interesting consequences, as we shall see.

In the above example, we have used `auto_ptr`s to provide automatic cleanup of heap based objects; the `auto_ptr` will always call the destructor of its body before it goes out of scope.

auto_ptr Ownership

- What happens when you copy **auto_ptr**s?
 - original **auto_ptr** releases body
 - internal pointer set to 0
 - new **auto_ptr** takes on ownership

```
void ChangeOwnership()
{
    // transfer ownership between auto_ptr
    auto_ptr<person> ap1(new person("John"));
    auto_ptr<person> ap2;

    ap2 = ap1;    // switch ownership (ap1 now empty)
    ap2->print();
} // ap2 cleans up
```

What happens when you copy **auto_ptr**s? The answer is that copying semantics are controlled by the ownership issue, namely that only one **auto_ptr** can own a given body. So when you copy an **auto_ptr** the new **auto_ptr** takes on ownership and as a consequence the original **auto_ptr** must release the body. When an **auto_ptr** releases a body its internal pointer is set to 0.

Of course when the new **auto_ptr** goes out of scope it will call the body's destructor and hence clean up.

Const auto_ptr

- **Ownership is permanent**
 - can't reassign const auto_ptr
 - can't copy the auto_ptr
 - can't release or reset
- **Compiler enforces ownership**
 - picks up silly mistakes

```
void ConstAutoPointers()
{
    // ownership permanent
    const auto_ptr<person> me(new person("Chris"));
    auto_ptr<person> other(new person("other"));

    me = other;                // can't reassign
    other = me;                // can't copy
    person* ptr = me.release(); // can't release
    me.reset();                 // can't reset
}
```

Copyright ©1994-2010 CRS Enterprises

119

An interesting variant with auto_ptr is a const auto_ptr. With a const auto_ptr ownership is permanent and that in turn means you

- can't reassign const auto_ptr
- can't copy the auto_ptr
- can't release or reset the auto_ptr

The compiler enforces these ownership rules.

Using auto_ptrs

- **auto_ptrs are very useful**
 - in many situations, but ...
- **semantics of ownership unusual**
 - auto_ptr becomes empty if copied
 - const auto_ptr can't be copied
- **copying semantics incompatible with STL**
 - STL relies on standard copying semantics
 - **don't use auto_ptr in STL containers**
 - and be careful with other software ...

Copyright ©1994-2010 CRS Enterprises

120

Obviously auto_ptrs are very useful in many situations, but you must be aware of the unusual ownership semantics. Remember that the auto_ptr becomes empty if copied, reset or released. A const auto_ptr can't be copied and represents a permanent ownership relationship with its body.

In particular, don't use an auto_ptr in STL containers because the auto_ptr copying semantics are incompatible with the STL. If you add an auto_ptr to a STL container be assured that disaster will ensue.

Boost Smart Pointers

scoped_ptr	<boost/scoped_ptr.hpp>	Simple sole ownership of single objects. Noncopyable.
scoped_array	<boost/scoped_array.hpp>	Simple sole ownership of arrays. Noncopyable.
shared_ptr	<boost/shared_ptr.hpp>	Object ownership shared among multiple pointers.
shared_array	<boost/shared_array.hpp>	Array ownership shared among multiple pointers
weak_ptr	<boost/weak_ptr.hpp>	Non-owning observers of an object owned by shared_ptr.
intrusive_ptr	<boost/intrusive_ptr.hpp>	Shared ownership of objects with an embedded reference count

The Boost smart pointer library provides six smart pointer class templates. These classes provide smart pointers for heap based objects and arrays. Some of the classes provide direct ownership of bodies, whilst others provide reference counting.

Scoped Pointer

- **simple smart pointer handle - body**
 - manages body by taking ownership
 - copying not allowed
- **used for automatic cleanup of heap based objects**
 - handle on stack uses RAII
- **can't be used with STL containers**
 - because of non copy restriction
- **alternative to `auto_ptr`**
 - but transfer of body ownership is not allowed

Copyright ©1994-2010 CRS Enterprises

122

The `scoped_ptr` is a simple smart pointer handle - body implementation where the handle manages the body by taking sole ownership. Unlike reference counted pointers, `scoped` pointers cannot be copied and die when they go out of scope.

The principal use of the `scoped_ptr` is for automatic cleanup of heap based objects using the Resource Acquisition is Initialization (RAII) idiom. This works because the handle is stack based and only the body is placed on the heap.

The non copy restriction implies that a `scoped_ptr` cannot be used with STL containers; use `shared_ptr` if that is your intention.

The `scoped_ptr` is an alternative to the STL `auto_ptr` and behaves similarly to a `const auto_ptr` in that transfer of body ownership is not allowed.

Scoped Array

- **simple smart pointer handle - body**
 - similar to `scoped_ptr`, but for arrays
- **alternative to using `std::vector`**
 - vector is much more flexible
 - but if its efficiency you are after ...
- **can't be used with STL containers**
 - because of non copy restriction

The `scoped_array` is used in the same way as `scoped_ptr` for heap based arrays. The STL already has the `vector` class for dynamic arrays and this should be preferred in most situations. However, the `scoped_array` is a very simple handle - body implementation and is therefore more efficient for fix sized arrays. So use `vector` unless efficiency is your primary concern.

Like the `scoped_ptr`, a `scoped_array` can't be copied and hence can't be used with STL containers.

Shared Pointer

- **reference counted pointer**
 - body is guaranteed to be deleted when the last `shared_ptr` pointing to it is destroyed
- **every `shared_ptr` is CopyConstructible and Assignable**
 - therefore can be used in standard library containers
- **with comparison operators**
 - `shared_ptr` works with the standard library's associative containers
- **be careful of cyclic references**
 - use `weak_ptr` to "break cycles"

The `shared_ptr` is a reference counted pointer whose body is guaranteed to be deleted when the last `shared_ptr` pointing to it is destroyed.

Shared pointers are copy constructible and assignable - conditions that must be met before a class can be used in the STL. Thus `shared_ptr` can be used with standard library containers.

Furthermore, `shared_ptr` has comparison operators which allows it to be used with the standard library's associative containers.

As with all reference counted artifacts, you must be careful of cyclic references. You can use `weak_ptr` to "break cycles".

Shared Array

- **reference counted pointer to an array**
 - body is guaranteed to be deleted when the last `shared_array` pointing to it is destroyed
- **like `shared_ptr`, these pointers can be used with**
 - standard library containers
 - standard library's associative containers
- **use `vector` ...**
 - unless efficiency is primary concern

The `shared_array` is used with arrays to provide a reference counted handle body combination. The pointer is provided as an efficient alternative to the `vector` class. It can be used with both standard library containers and associative containers.

Weak Pointer ...

- **stores a weak reference to an object that's already managed by a `shared_ptr`**
- **to access the object, a `weak_ptr` can be converted to a `shared_ptr`**
 - using the `shared_ptr` constructor or the member function `lock`
- **when the last `shared_ptr` to the object goes away the object is deleted**
 - attempts to obtain a `shared_ptr` from the `weak_ptr` will fail
 - constructor throws an exception `boost::bad_weak_ptr`
 - `weak_ptr::lock` returns an *empty* `shared_ptr`
- **can be used with standard library containers and associative containers.**
- **`weak_ptr` operations never throw exceptions**

Copyright ©1994-2010 CRS Enterprises

126

The `weak_ptr` stores a weak reference to an object that's already managed by a `shared_ptr`. Weak references are used to break reference counted cycles.

Weak pointers can't access their body directly. To access the body, a `weak_ptr` can be converted to a `shared_ptr` using the `shared_ptr` constructor or the member function `lock`.

When the last `shared_ptr` to the object goes away the object is deleted, attempts to obtain a `shared_ptr` from the `weak_ptr` will fail. In the case of the constructor a `boost::bad_weak_ptr` exception is thrown and in the case of `lock()` an empty `shared_ptr` is returned.

Weak pointers can be used with standard library containers and associative containers.

Weak pointers operations never throw exceptions.

... Weak Pointer

- **often dangerous in multithreaded programs**
 - other threads can delete body without the weak pointer noticing
 - a dangling pointer

```
shared_ptr<int> p(new int(5));  
weak_ptr<int> q(p);
```

create a temporary
shared_ptr from q

```
// some time later
```

```
if(shared_ptr<int> r = q.lock())  
{  
    // use *r  
}
```

r holds a reference body. Reference count
can't go to zero until were finished

Be careful with `weak_ptr` in multi-threaded programs. You need to guard against other threads releasing their `shared_ptr`s and the body getting deleted. If that happens they `weak_ptr` will become a dangling pointer.

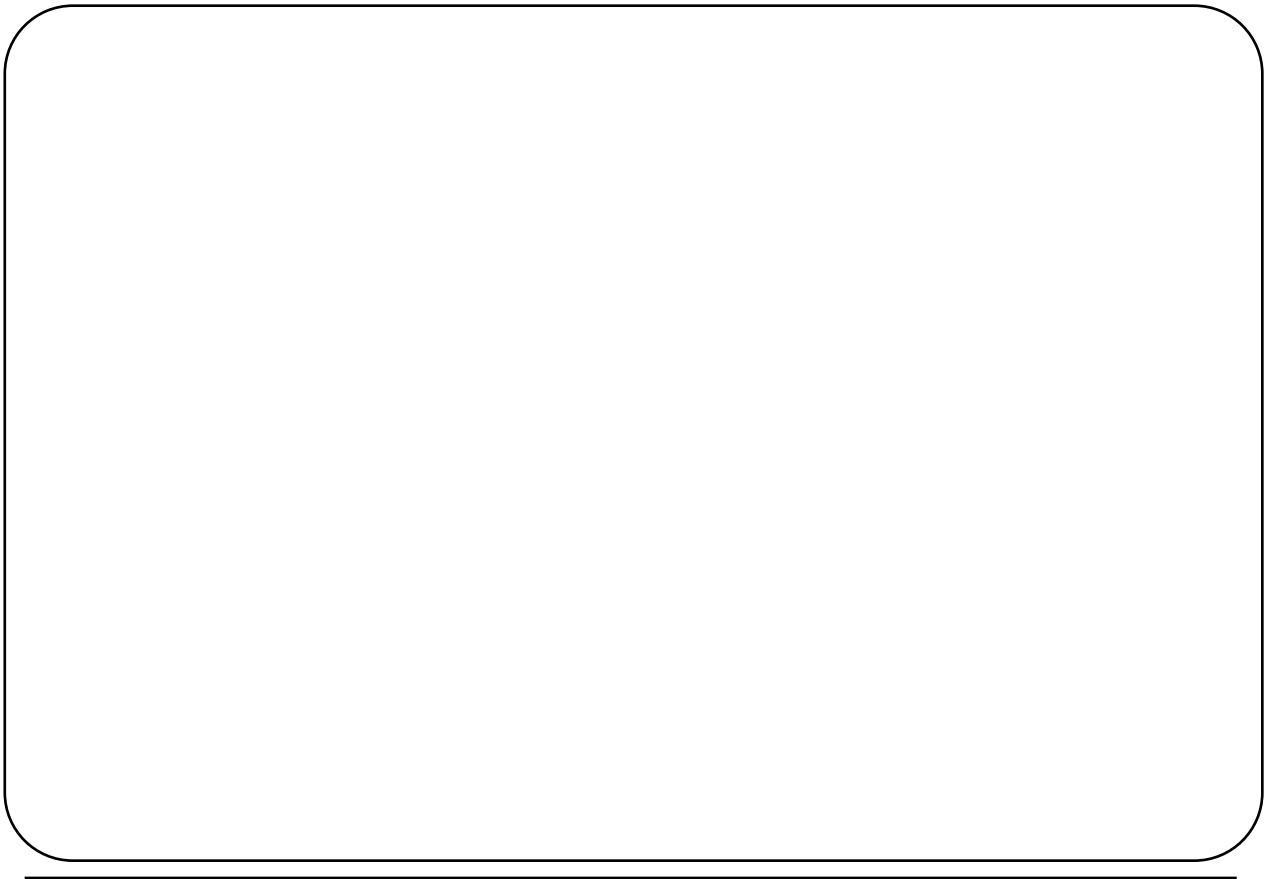
To avoid this situation, create a temporary shared pointer from the wak pointer before you use it, effectively incrementing the reference count on the body by one and preventing the body being deleted prematurely.

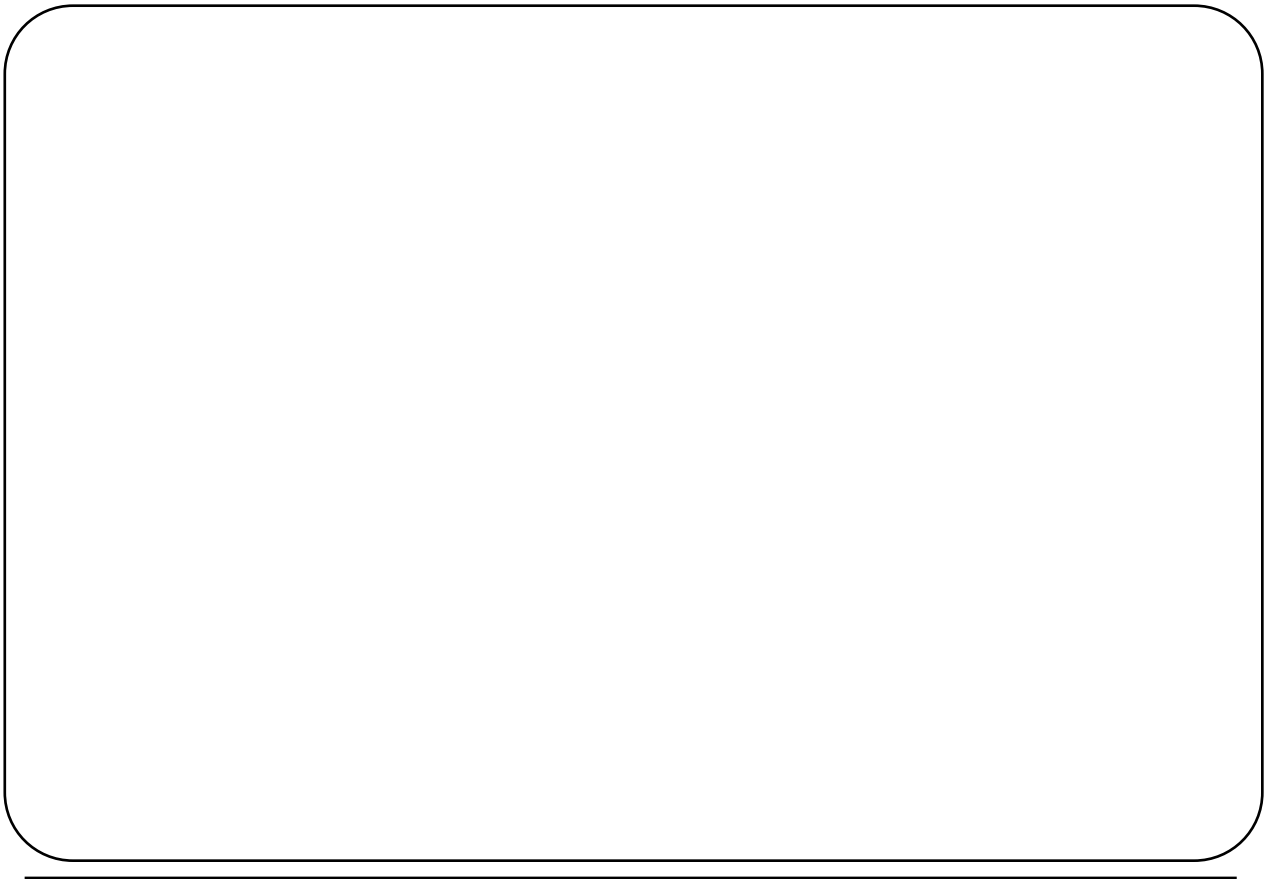
Intrusive Pointer ...

- **intrusive_ptr stores a pointer to a body with an embedded reference count**
 - every intrusive_ptr increments/decrements the reference count by using calls to intrusive_ptr_add_ref/intrusive_ptr_release
 - user is expected to provide suitable definitions of these two functions
- **intended for frameworks that provide reference counted class**
 - usually you should prefer shared_ptr

An intrusive_ptr stores a pointer to a body with an embedded reference count. The intrusive_ptr increments the reference count by using calls to intrusive_ptr_add_ref and decrements the reference count by using calls to intrusive_ptr_release.

These smart pointers are provided for compatibility with frameworks that already provide reference counted classes. Usually you should prefer to use shared_ptr.





Chapter 10

10

Boost

- **What is Boost**
 - visit home page
 - Some of the Libraries ...
- **Any** - variant data types
- **Assign** - simplifying assignment
- **Bind** - generalised callbacks
- **Function** - generalised callbacks
- **Regex** - pattern matching
- **Spirit** - mini parsers
- **uBLAS** - vectors and matrices



Chapter 10

What is Boost

- **Free peer-reviewed portable C++ source libraries**
 - license is very similar to the BSD license and the MIT license
 - written by professional C++ programmers
 - work well with the C++ Standard Library
 - in regular use by thousands of programmers
- **Boost libraries are suitable for standardization**
 - Many Boost libraries are part of the C++ 2011 Standard
 - many other useful libraries
- **Template Based**
 - most of the Boost libraries are distributed as templates
 - some compile to shared objects or DLLs

Copyright ©1994-2010 CRS Enterprises

133

Boost provides free peer-reviewed portable C++ source libraries. The emphasis is on libraries which work well with the C++ Standard Library. The libraries are intended to be useful across a broad spectrum of applications and are in regular use by thousands of programmers.

A further goal is to establish "existing practice" and provide reference implementations so that Boost libraries are suitable for eventual standardization. Several Boost libraries will be included in the C++ 2011 Standard Library.

Although Boost was begun by members of the C++ Standards Committee Library Working Group, participation has expanded to include thousands of programmers from the C++ community at large.

Many of the Boost libraries are pure templates and therefore only require a compile step. However, a few of the libraries do need to be built into shared objects or DLLs.

Boost Web Site

- Visit the Boost home page
 - <http://www.boost.org/>



Copyright ©1994-2010 CRS Enterprises

134

The Boost web site provides free peer-reviewed portable C++ source libraries. Downloads are available for all libraries, and for those requiring shared objects (or DLLs) a build facility is provided.

Complete documentation, FAQ and many tutorials are available on the site, plus links to mail groups and support.

Just Some of the Libraries ...

- **Any**
 - is a variant value type supporting copying of any value type and safe checked extraction of that value strictly against its type
- **Assign**
 - makes it easy to fill containers with lists of data by overloading `operator,()` and `operator>()()`
- **Bind and Function**
 - generalization of the callback mechanism supporting arbitrary function objects, functions, function pointers, and member function pointers
- **Regex**
 - regular expression engine integrates with STL string class
- **Spirit**
 - recursive-descent parser generator framework implemented using template meta-programming; enables a target grammar to be written exclusively in C++
- **uBLAS**
 - templated C++ classes for dense, unit and sparse vectors, dense, identity, triangular, banded, symmetric, hermitian and sparse matrices

Copyright ©1994-2010 CRS Enterprises

135

The complete Boost library is too extensive to be covered in this chapter. Instead, we will investigate examples from the above libraries in subsequent pages.

Any

```
#include <boost/any.hpp>
using namespace std;

typedef std::list<boost::any> LIST_OF_ANY;
using boost::any_cast;

int main()
{
    // sample data
    int i = 100;
    double d = 4.53;
    string myString("mystring");
    char* myCharArray = "charArray";

    // build list of mixed types
    LIST_OF_ANY myList;
    myList.push_back(i);
    myList.push_back(d);
    myList.push_back(myString);
    myList.push_back(myCharArray);

    // iterate through list to determine type
    for_each(myList.begin(),
             myList.end(),
             DetermineType);
}

void DetermineType(boost::any& operand)
{
    try
    {
        if(any_cast<int>(operand))
            cout << "int" << endl;
    }
    catch(const boost::bad_any_cast &) {}

    if(operand.type() == typeid(double))
        cout << "double" << endl;
    if(operand.type() == typeid(string))
        cout << "string" << endl;
    if(operand.type() == typeid(char*))
        cout << "char*" << endl;
}
```

Copyright ©1994-2010 CRS Enterprises

136

The idea behind this library is to be able to compose lists of variant value types such that type information is retained. When retrieving an Any object from a container the type information can be inferred using a cast

```
if(any_cast<int>(operand))
```

where a `boost::bad_any_cast` exception is thrown if the cast fails.

Alternatively you can use the `boost::Any::type()` method

```
if(operand.type() == typeid(double))
```


Assign - Simple Example

- **Overload +=**
 - to make it easy to add to the container
- **Overload comma**
 - to make it easy to form a list

```
#include <boost/assign/std/vector.hpp>
#include <boost/assert.hpp>
#include <iostream>
using namespace std;
using namespace boost::assign;

void Print(const int& i)
{
    cout << i; cout.flush();
}

int main()
{
    vector<int> values;
    // insert values at the end of the container
    values += 1,2,3,4,5,6,7,8,9;

    for_each(values.begin(), values.end(), Print);
}
```

123456789

This library makes it easy to form lists and add them to a container. It allows a Python like syntax by overloading the += and the comma operator.

Assign - More Advanced

- **Overload ()**
 - easy to define name-value pairs

```
int main()
{
    map<string,int> months;
    insert( months )
        ( "january", 31 )( "february", 28 )
        ( "march", 31 )( "april", 30 )
        ( "may", 31 )( "june", 30 )
        ( "july", 31 )( "august", 31 )
        ( "september", 30 )( "october", 31 )
        ( "november", 30 )( "december", 31 );

    BOOST_ASSERT( months.size() == 12 );
    BOOST_ASSERT( months["january"] == 31 );

    for_each(months.begin(), months.end(), Print);
}
```

```
#include <boost/assign/list_inserter.hpp>
#include <boost/assert.hpp>
#include <iostream>
#include <string>
#include <map>
using namespace std;
using namespace boost::assign;

void Print(const pair<string,int>& mypair)
{
    cout << mypair.first << ", "
          << mypair.second
          << endl;
}
```

```
april,30
august,31
december,31
february,28
january,31
july,31
june,30
march,31
may,31
november,30
october,31
september,30
```

Here is a more interesting example from the Assign library. The insert method is used with the overloaded operator()() to simplify adding a set of name-value pairs to a map container.

BOOST_ASSERT is a general facility and not part of the Assign library.

Bind and Function ...

- **Bind** library creates Function Objects
 - generalization of the `std::bind1st` and `std::bind2nd`
 - supports arbitrary function objects, functions, function pointers, and member function pointers
 - able to bind any argument to a specific value
 - or route input arguments into arbitrary positions
 - support placeholders `_1, _2, _3, ... _9`
- **Function** library contains templates that are function object wrappers
 - similar to a generalized callback
 - often used with Bind

... Bind and Function

- Note template parameter
 - when creating binder

```
int main()
{
    // create binder
    boost::function<int(int)> binder;

    // bind to a global function
    binder = &DoubleIt;

    // invoke binder
    int result = binder(5);

    // try it with objects and member functions
    A a(100);
    binder = boost::bind(&A::Add,a,_1);
    result = binder(2);
}
```

```
#include <vector>
#include <iostream>
#include <boost/bind.hpp>
#include <boost/function.hpp>
#include <boost/assign/std/vector.hpp>
using namespace std;
using namespace boost::assign;

class A
{
private:
    int x;
public:
    A(int x) : x(x) {}
    int Add(int y) { return x + y; }
};

int DoubleIt(int a) { return 2 * a; }
```

_1 is parameter 1

Copyright ©1994-2010 CRS Enterprises

140

This is an extensive library that is a generalization of the C++ callback mechanism. The library supports arbitrary function objects, functions, function pointers, and member function pointers. Examples of binding to functions and member function pointers are given above.

The callback objects are called binders and are created using the `boost::function` template. Note that in the example shown, the template takes a single parameter that defines the signature of the function being bound

```
int DoubleIt(int)
```

```
int A::Add(int)
```

Lambda ...

- Creates anonymous functions

```
typedef boost::function<int(int,int,int)>  
FP;
```

```
int main()  
{
```

```
    FP fp = (_1 + _2) * _3;
```

behaves as if you create a function

```
    // call the function
```

```
    int result = fp(2, 4, 10);  
    cout << result << endl;
```

```
}
```

```
int lambda_function(int _1, int _2, int _3)  
{  
    return (_1 + _2) * _3;  
}
```

... Lambda

```
typedef boost::function<string(const string&, const string&)> FP;
class Button
{
public:
    void onPress(const string& name, const FP& callbackFunction)
    {
        this->name = name;
        this->callbackFunction = callbackFunction;
    }
    string press()
    {
        return callbackFunction(name, string(" pressed "));
    }
private:
    string name;
    FP callbackFunction;
};
```




Diagram illustrating the lambda function call in the `press()` method. The variable `_1` points to the `name` parameter, and the variable `_2` points to the `string(" pressed ")` argument.

```
Button button;
button.onPress("button 2", ret<string>(_2 + _1));
cout << button2.press() << endl;
```

Regex

```
#include <string>
#include <iostream>
#include <iterator>
#include <boost/regex.hpp>
#include <boost/algorithm/string/regex.hpp>
using namespace std;
using namespace boost;

int main()
{
    string str1("abc_(456)_123_(123)_cde");

    // Replace all substrings matching (digit+)
    replace_all_regex_copy( str1, regex("\\([0-9]+\\)"), string("#$1#") )

    // Erase all substrings matching (letter+)
    erase_all_regex_copy( str1, regex("[[:alpha:]]+") )

    // in-place regex transformation
    replace_all_regex( str1, regex("_\\([^\n\\)]*\\)"), string("-$1-") );
}
```

```
abc_#456#_123_#123#_cde
_(456)_123_(123)_
abc_-(456)-_123_-(123)-_cde
```

Copyright ©1994-2010 CRS Enterprises

143

Regular Expressions are a form of pattern-matching that are often used in text processing. Many users will be familiar with the Unix utilities `grep`, `sed` and `awk`, and the programming language `Perl`, each of which make extensive use of regular expressions. Traditionally C++ users have been limited to the POSIX C API's for manipulating regular expressions, but these API's do not represent the best way to use the library. That's why the Boost library is needed.

The class `boost::basic::regex` is the key class in this library; it represents a "machine readable" regular expression, and is very closely modelled on `std::basic_string`. The library mimics the way regular expression patterns are defined in `Perl`.

Spirit Parser

- implemented using Expression Templates

- define your own grammar using 'EBNF' syntax
- parser overloads C++ operators to get close to EBNF

```

template <typename Iterator>
bool parse_numbers(Iterator first, Iterator last) {
    using qi::double_;
    using qi::phrase_parse;
    using ascii::space;

    bool r = phrase_parse(
        first,                      /*< start iterator >*/
        last,                      /*< end iterator >*/
        double_ >> *(',') >> double_, /*< the parser >*/
        space                     /*< the skip-parser >*/
    );
    if (first != last) return false; // fail if we did not get a match
    return r;
}

```

did it parse?

grammar for a comma separated list of real numbers

token

Copyright ©1994-2010 CRS Enterprises

144

The Spirit Parser uses Expression Templates to allow you to define your own grammar (mini parser) which you can embed in your C++ code. The parser is meant to approximate Extended Backus-Normal Form.

Spirit makes extensive use of operator overloading to enable your grammar to approximate EBNF. The advantage of this approach is that the grammar is an integral part of the C++ application (indeed it is written in C++) and not a separate module. This makes it ideal for simple parsers, but it can be used for really complex tasks.

In the above example (taken from the Spirit documentation) a parser is written to accept a list of real numbers (comma separated, interspersed with white space).

Tuples

- Tuple (or n -tuple) is a fixed size collection of elements

```
int main( )
{
    // creation
    tuple<int, double, string, Point> myTuple(1, 3.14, string("Hello"), Point(3,4));

    // printing
    cout << myTuple << endl;

    // make_tuple
    doit(2, 1.72, "Goodbye", Point(10,50));

    // extraction
    int i = myTuple.get<0>();      cout << i << endl;
    double d = myTuple.get<1>();   cout << d << endl;
    string s = myTuple.get<2>();   cout << s << endl;
    Point p = myTuple.get<3>();    cout << p << endl;
}
```

Conversion

- Using cast like syntax

```
int main()
{
    string s1 = boost::lexical_cast<string> (2);
    string s2 = boost::lexical_cast<string> (8);
    cout << s1 + s2 << endl;

    try
    {
        double x1 = boost::lexical_cast<double> ("2.54");
        double x2 = boost::lexical_cast<double> ("2???54");
    }
    catch(boost::bad_lexical_cast& e)
    {
        cout << "Exception caught - " << e.what() << endl;
    }
}
```

uBLAS - Vectors

- Linear Algebra

– vectors, matrices ...

```
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/io.hpp>
```

```
int main () {
    using namespace std;
    using namespace boost::numeric::ublas;
    using boost::numeric::ublas::vector;
```

```
    catch(bad_index ex)
    {
        cout << ex.what() << endl;
    }
    catch(...)
    {
        cout << "unknown error" << endl;
    }
}
```

```
vector<int> v1(3);
vector<int> v2(3);
matrix<int> m(3, 3);
```

```
try {
    v1(0) = 2;
    v1(1) = 4;
    v1(2) = 1;
    v2(0) = 2;
    v2(1) = 0;
    v2(2) = 3;
```

```
    int dot_product = inner_prod(v1,v2);
    cout << cross_product << endl;
    m = outer_prod(v1, v2);
    cout << m << endl;
}
```

7
[3,3]((4,0,6),(8,0,12),(2,0,3))

Copyright ©1994-2010 CRS Enterprises

147

The uBLAS library contains classes for linear algebra. Some of the simpler classes deal with vectors and matrices. In this example, the inner and outer product of two vectors is taken.

uBLAS - Matrices

```
int main ()
{
    using namespace boost::numeric::ublas;
    matrix<double> m1(2, 3);
    matrix<double> m2(3, 2);
    matrix<double> m(2, 2);

    try {
        m1(0,0) = 2;
        m1(0,1) = 1;
        m1(0,2) = 2;
        m1(1,0) = 4;
        m1(1,1) = 3;
        m1(1,2) = 5;
        m2(0,0) = 2;
        m2(0,1) = 3;
        m2(1,0) = 1;
        m2(1,1) = 6;
        m2(2,0) = 2;
        m2(2,1) = 7;

        m = prod(m1,m2);
        cout << m << endl;
    }
    catch(bad_index ex)
    {
        cout << ex.what() << endl;
    }
    catch(...)
    {
        cout << "unknown error" << endl;
    }
}
```

[2,2]((9,26),(21,65))

Copyright ©1994-2010 CRS Enterprises

148

This example uses two dimensional matrix operations. A 2 x 3 matrix is multiplied with a 3 x 2 matrix to give a square 2 x 2 matrix.

Library Versions

- **Debug and Release versions available**
 - dynamic and static linking
- **Naming Conventions**
 - **mt** multi-threaded library
 - **s** statically linked
 - **d** no optimization or inlining, full debug symbols enabled

```

C:\boost_1_46_1\bin.v2\libs\regex\build>find . -name '*.a'
./gcc-mingw-4.5.2/debug/libboost_regex-mgw45-d-1_46_1.a
./gcc-mingw-4.5.2/debug/link-static/libboost_regex-mgw45-d-1_46_1.a
./gcc-mingw-4.5.2/debug/link-static/runtime-link-static/libboost_regex-mgw45-sd-1_46_1.a
./gcc-mingw-4.5.2/debug/link-static/runtime-link-static/threading-multi/libboost_regex-mgw45-mt-sd-1_46_1.a
./gcc-mingw-4.5.2/debug/link-static/threading-multi/libboost_regex-mgw45-mt-d-1_46_1.a
./gcc-mingw-4.5.2/debug/threading-multi/libboost_regex-mgw45-mt-d-1_46_1.dll.a
./gcc-mingw-4.5.2/release/libboost_regex-mgw45-1_46_1.dll.a
./gcc-mingw-4.5.2/release/link-static/libboost_regex-mgw45-1_46_1.a
./gcc-mingw-4.5.2/release/link-static/runtime-link-static/libboost_regex-mgw45-s-1_46_1.a
./gcc-mingw-4.5.2/release/link-static/threading-multi/libboost_regex-mgw45-mt-s-1_46_1.a
./gcc-mingw-4.5.2/release/link-static/threading-multi/libboost_regex-mgw45-mt-1_46_1.a
./gcc-mingw-4.5.2/release/threading-multi/libboost_regex-mgw45-mt-1_46_1.dll.a

C:\boost_1_46_1\bin.v2\libs\thread\build>gcc-mingw-4.5.2>find . -name '*.a'
./debug/link-static/runtime-link-static/threading-multi/libboost_thread-mgw45-mt-sd-1_46_1.a
./debug/link-static/threading-multi/libboost_thread-mgw45-mt-d-1_46_1.a
./debug/threading-multi/libboost_thread-mgw45-mt-d-1_46_1.dll.a
./release/link-static/runtime-link-static/threading-multi/libboost_thread-mgw45-mt-s-1_46_1.a
./release/link-static/threading-multi/libboost_thread-mgw45-mt-1_46_1.a
./release/threading-multi/libboost_thread-mgw45-mt-1_46_1.dll.a
  
```

Copyright ©1994-2010 CRS Enterprises

149

In order to choose the right binary for your build configuration you need to know how Boost binaries are named. Each library filename is composed of a common sequence of elements that describe how it was built. For example, `libboost_regex-vc71-mt-d-1_34.lib` can be broken down into the following elements:

`lib`

Prefix: except on Microsoft Windows, every Boost library name begins with this string. On Windows, only ordinary static libraries use the `lib` prefix; import libraries and DLLs do not.

`boost_regex`

Library name: all boost library filenames begin with `boost_`.

`-vc71`

Toolset tag: identifies the toolset and version used to build the binary.

`-mt`

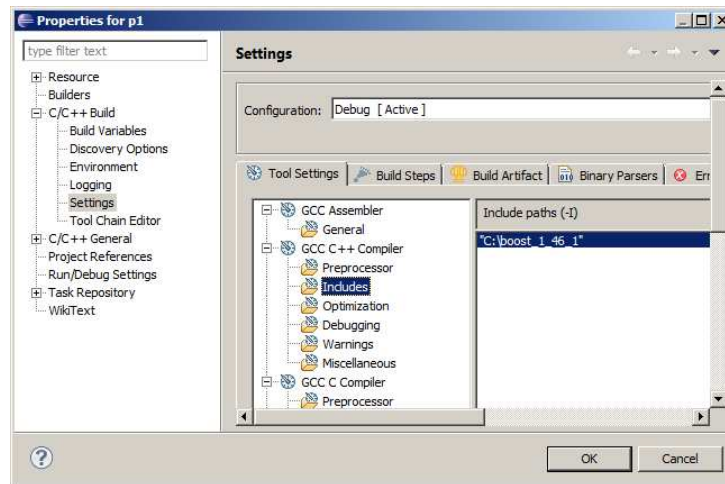
Threading tag: indicates that the library was built with multithreading support enabled. Libraries built without multithreading support can be identified by the absence of `-mt`.

`-d`

ABI tag: encodes details that affect the library's interoperability with other compiled code.

Compiling

- **Project properties**
 - set the include paths



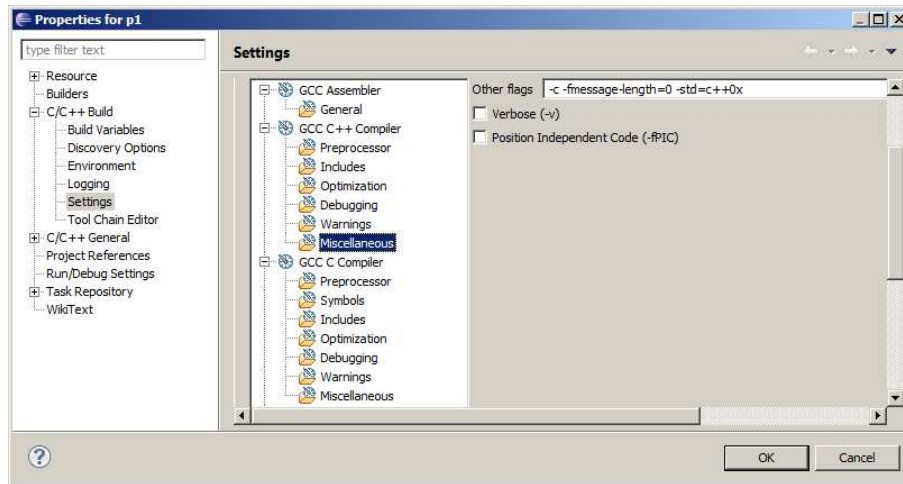
Copyright ©1994-2010 CRS Enterprises

150

The settings dialog allows you to set various compiler options. In this instance we are specifying the location of the boost header files (same as the boost installation directory).

Compiling for C++ 2011

- Project properties
 - set miscellaneous options (**-std=c++0x**)



Copyright ©1994-2010 CRS Enterprises

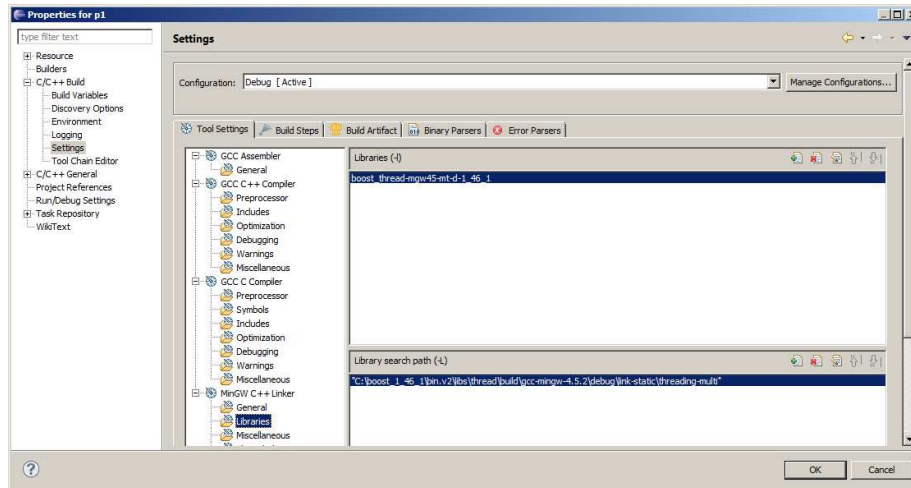
151

If you want to use C++ 2011 features that have been implemented in GCC then add the following to the miscellaneous setting:

`-std=c++0x`

Linking

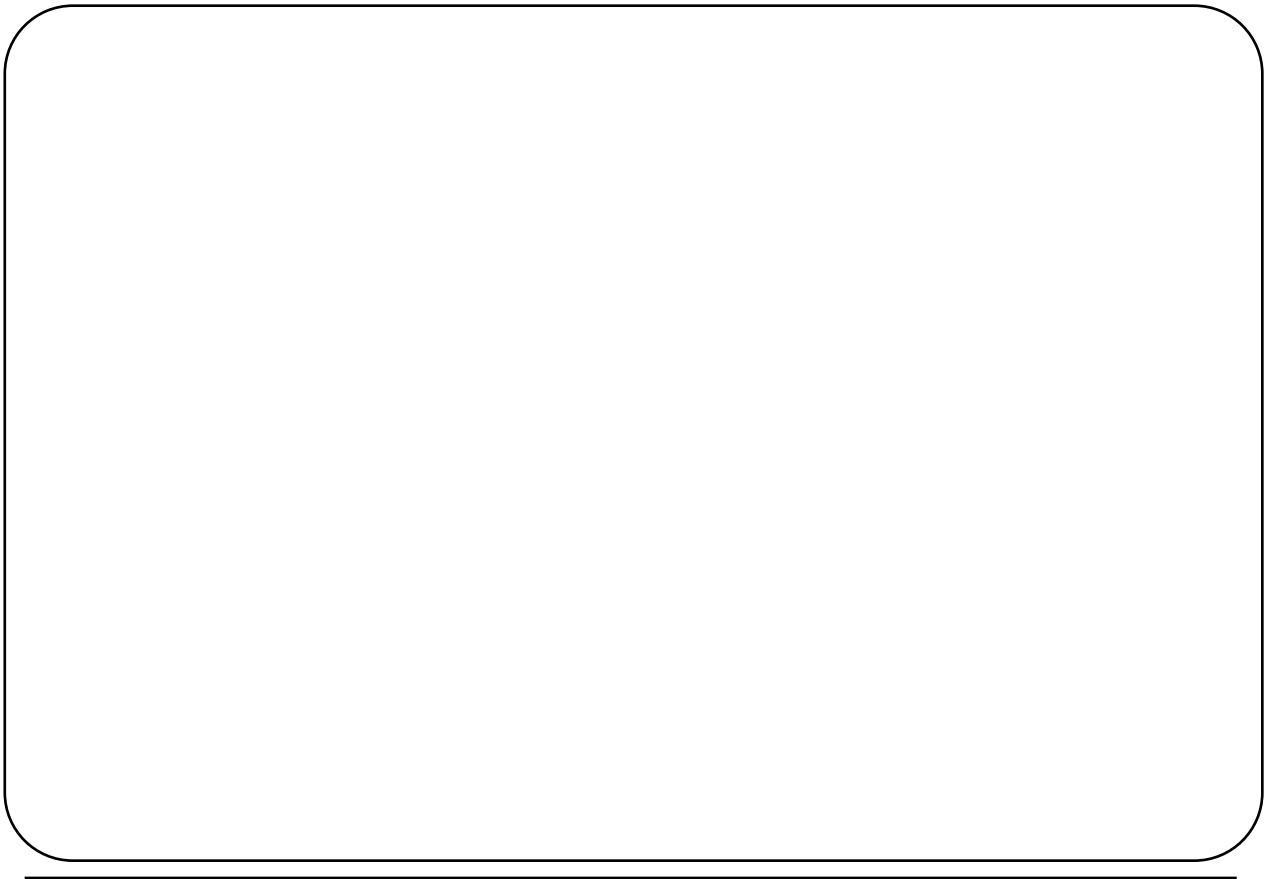
- **Project properties**
 - set library options (library and path)



Copyright ©1994-2010 CRS Enterprises

152

For libraries, you must specify both the library to be used and the full path to the library. Use the MinGW C++ Linker settings to do this.



Chapter 11

11

Copyright ©1994-2010 CRS Enterprises

154

Object Oriented Callbacks

- **C++'s Callback Operators**

– `.->.*->*`

– binding objects to methods

- **Command Pattern**

– basic pattern

– using adaptors

- **Further Concepts**

– using inheritance

– Boost library



Chapter 11

Traditional Callbacks

- **The C callback model uses function pointers**
 - functions are stateless
 - so functions requiring a context need data passed in separately
- **The C callback function model is too primitive**
 - functions are defined at compile time
 - not dynamic
 - error prone
 - sometimes typeless
 - unsafe
 - inextensible

Copyright ©1994-2010 CRS Enterprises

156

Event driven programming, such as handling a button press in a GUI system or receiving an interrupt from another thread, is characterised by the use of callbacks. In C this is implemented using global function pointers.

Unfortunately, global function pointers do not work well in an object oriented environment and it is fair to say that the C callback function model is too primitive. Member functions can't be used directly and global functions often require a context to simulate member functions. The context data is passed in separately, sometimes lacking type information (i.e. void*), leading to an error prone and unsafe system.

Multiple roots of control, association of data with function, grouping of related functions, generic interfaces with alternative implementations, etc. are mapped more easily into an object model than into a procedural one.

C++'s Callback Operators

- 4 Object-Method Binding Operators

- object bound
 - at compile time or runtime
- method bound
 - at compile time or runtime

		<i>object</i>	
		compile time	run time
<i>method</i>	compile time	.	->
	run time	.*	->*

Copyright ©1994-2010 CRS Enterprises

157

We are all familiar with the

.

->

operators. These operators bind an object to a method; the dot operator binds the object at compile time and the arrow operator binds at runtime. In both cases the method is bound at compile time.

Two lesser known operators

.*

->

are used to bind a method at runtime. Using all 4 operators allows either the object or the method to be bound at compile time or run time.

As we will see, these operators are C++'s way of implementing object oriented callbacks.

Object Oriented Callbacks(1)

- Bind object to method at runtime

```
void Callback::Load(PTR_TO_MEMBER pFunction, A& object)
{
    pf = pFunction;
    po = &object;
}
```

```
void Callback::Fire(int n)
{
    (po->*pf)(n);
}
```

```
class Callback
{
public:
    typedef void (A::*PTR_TO_MEMBER)(int);
public:
    void Load(PTR_TO_MEMBER pFunction, A& object);
    void Fire(int);
private:
    A* po;
    PTR_TO_MEMBER pf;
};
```

Copyright ©1994-2010 CRS Enterprises

158

In this example, we intend to make Object Oriented callbacks to different methods for various objects. We select the method and the object sometime before the actual callback.

The Callback class defines Load and Fire methods. The Load method records the object and the method, but doesn't actually make the call. The call is not made until the Fire method is invoked.

Object Oriented Callbacks(2)

- **Fire**

- with different objects
- with different methods

```
int main()
{
    A a1(100);
    A a2(200);
    Callback c;

    c.Load(&A::Subtract, a1);
    c.Fire(25);
    c.Fire(40);

    c.Load(&A::Multiply, a2);
    c.Fire(2);
    c.Fire(3);
}
```

```
class A
{
public:
    A(int x0) : x(x0) {}
    virtual void Add(int n) { ... }
    virtual void Subtract(int n) { ... }
    virtual void Multiply(int n) { ... }
private:
    int x;
};
```

Copyright ©1994-2010 CRS Enterprises

159

To Load the callback select a pointer to a method and an object. The callback is executed by the Fire method. Notice that since each callback method takes a single int parameter; this parameter is passed to Fire and the callback is implemented using (see previous slide)

```
(po->*pf)(n);
```

Note that the syntax of a pointer to a member function requires use of the & operator; omitting the & will cause a compilation error.

The Command Pattern

- **Decouple request for an operation from its execution**

- load a request
- fire a request
 - using an execute() method

```
class Command
{
public:
    virtual void execute() = 0;
};

class Timer
{
private:
    vector<Command*> events;
    vector<Time> times;
public:
    void load_event(Time when, Command* what)
    {
        events.push_back(what);
        times.push_back(when);
    }

    void fire_events()
    {
        for(int i = 0; i < (int)events.size(); i++)
        {
            if(times[i] == ticks) events[i]->execute();
        }
    }
};
```

Copyright ©1994-2010 CRS Enterprises

160

The Command pattern is designed to decouple a request for an operation from its execution. An operation is defined in terms of an object bound to a method; a separate class is used to record this information.

In the above example, the load_event method records several operations in a vector along with the times when the callback is to be made. The fire_events method decides whether or not to execute the request.

Use Objects to Encapsulate Behaviour

- each event
 - encapsulated in a different class
 - class **IS A** Command
 - execute method has a different implementation

```
class HeatingOff : public Command
{
public:
    HeatingOff(Heating* t) : target(t) {}

    virtual void execute()
    {
        target->turn_off();
    }
private:
    Heating* target;
};
```

```
class HeatingOn : public Command
{
public:
    HeatingOn(Heating* t) : target(t) {}

    virtual void execute()
    {
        target->turn_on();
    }
private:
    Heating* target;
};
```

Copyright ©1994-2010 CRS Enterprises

161

If a system is to model several different events then the simplest approach is to define a separate class for each event. Of course the class must implement the Command interface.

The above system defines two events: turn_off and turn_on a heating system. These events are modelled by the HeatingOff and HeatingOn classes respectively.

Using the Command Pattern

```
int main()
{
    Time t4(4);
    Time t8(8);
    Heating system;
    Timer    controller;

    HeatingOn on(&system);
    controller.set(t4, &on);

    HeatingOff off(&system);
    controller.set(t8, &off);

    cout << "start of simulation";

    for(int i = 0; i < 12; i++)
    {
        controller.tick();
    }
}
```

```
class Timer
{
public:
    Timer() : ticks(0) {}
    void set( ... ) { ... }
    void tick() {
        ...
        events[i]->execute();
        ...
    }
};
```

- **Timer class**
 - performs the callbacks

162

The system can now be completed as shown above. The main programme creates the event objects and a Timer object is used to record the events and timings. The simulation sets off the timer and events are fired at t4 and t8.

Command Adaptors

- define one class for all events
 - neater than one class per event

```

class CommandAdapter : public Command
{
public:
    typedef void (Heating::*CALLBACK)();
    typedef Heating* DATA;

    CommandAdapter(CALLBACK c, DATA d) : callback(c), data(d) {}

    virtual void operator()()
    {
        (data->*callback)();
    }

private:
    CALLBACK callback;
    DATA data;
};

class Heating
{
public:
    void turn_on() { cout << "turn on"; }
    void turn_off() { cout << "turn off"; }
    void service() { cout << "service"; }
};
  
```

Copyright ©1994-2010 CRS Enterprises

163

The previous example shows how the command pattern can be used to good effect, but it is unfortunate that each event is represented by a separate class. Obviously, in some situations this could lead to an explosion of event classes.

A refinement of this pattern is to use a CommandAdaptor which dynamically selects the action and thereby removes the necessity for having a separate class per event.

This is achieved by defining a Heating class for each event represented as a separate method, rather than having separate HeatingOff, HeatingOn and HeatingService classes. The CommandAdaptor constructor records the Heating object and the action (on, off or service) and the callback is made with either an execute() method or with the operator()() method. It has become a tradition in C++ to prefer the overloaded () for the command pattern.

Command Adaptors in Action

```
int main()
{
    Time t4(4);
    Time t8(8);
    Time t10(10);
    Heating system;
    Timer controller;

    CommandAdapter on(&Heating::turn_on, &system);
    CommandAdapter off(&Heating::turn_off, &system);
    CommandAdapter service(&Heating::service, &system);

    controller.set(t4, &on);
    controller.set(t8, &off);
    controller.set(t10, &service);

    cout << "start of simulation";
    for(int i = 0; i < 12; i++)
    {
        controller.tick();
    }
}
```

```
class CommandAdapter : public Command
{
    ...
    virtual void operator()()
    {
        (data->*callback)();
    }
    ...
};
```

- **Fully dynamic**

- Heating system selected at runtime
- Event selected at runtime

Copyright ©1994-2010 CRS Enterprises

164

Now we can define separate CommandAdaptor objects to represent the callbacks.

The object on represents a callback on the object system with the method Heating::turn_on()

The object off represents a callback on the object system with the method Heating::turn_off()

The object service represents a callback on the object system with the method Heating::turn_service()

Clearly we can introduce further Heating objects if required.

Further Concepts

- **Using Inheritance**
 - C++ callback method supports polymorphism
 - declare a pointer to a Base member
 - it can point to a Derived member
- **Using Boost**
 - the Boost libraries provide extensive support for callbacks
 - **boost::bind**
 - encapsulates the .-> operator

Copyright ©1994-2010 CRS Enterprises

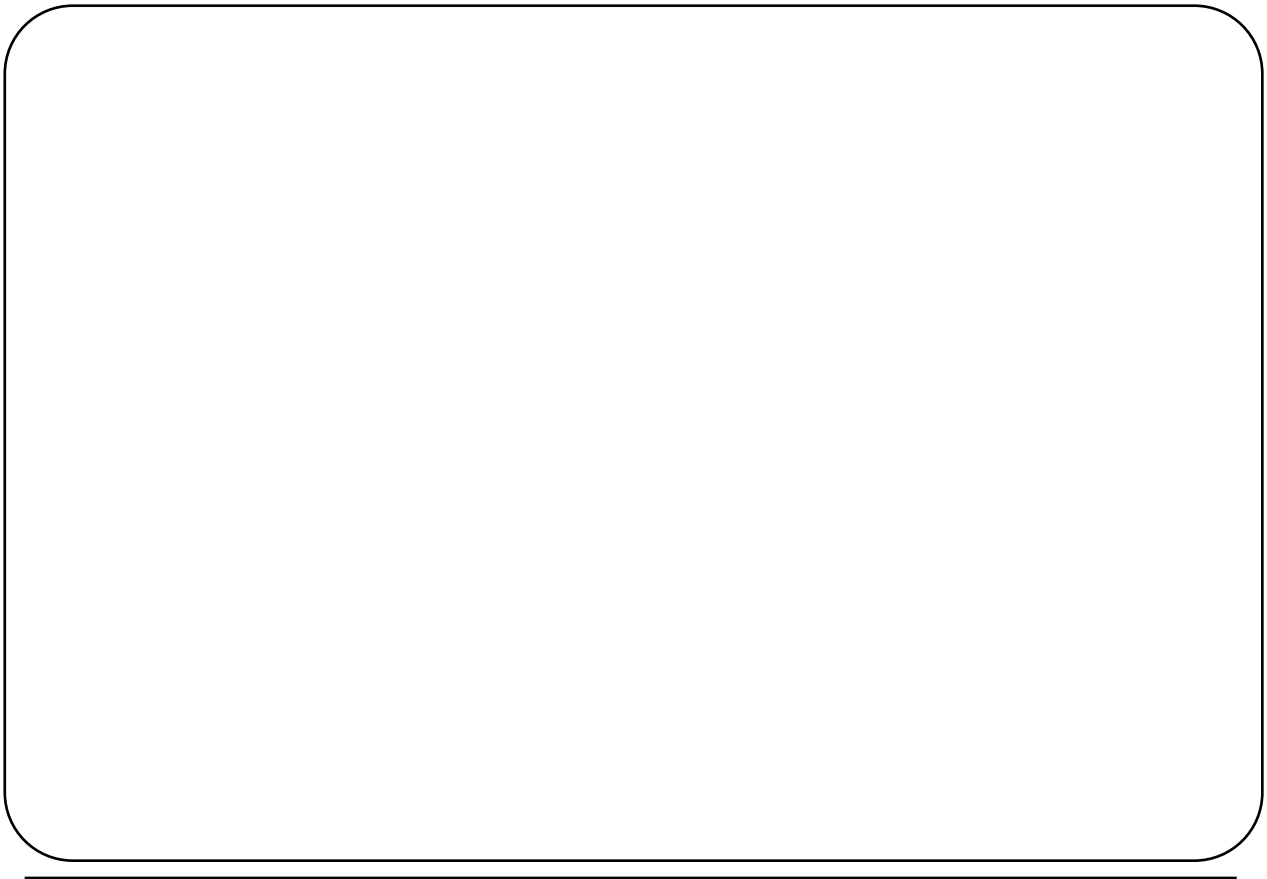
165

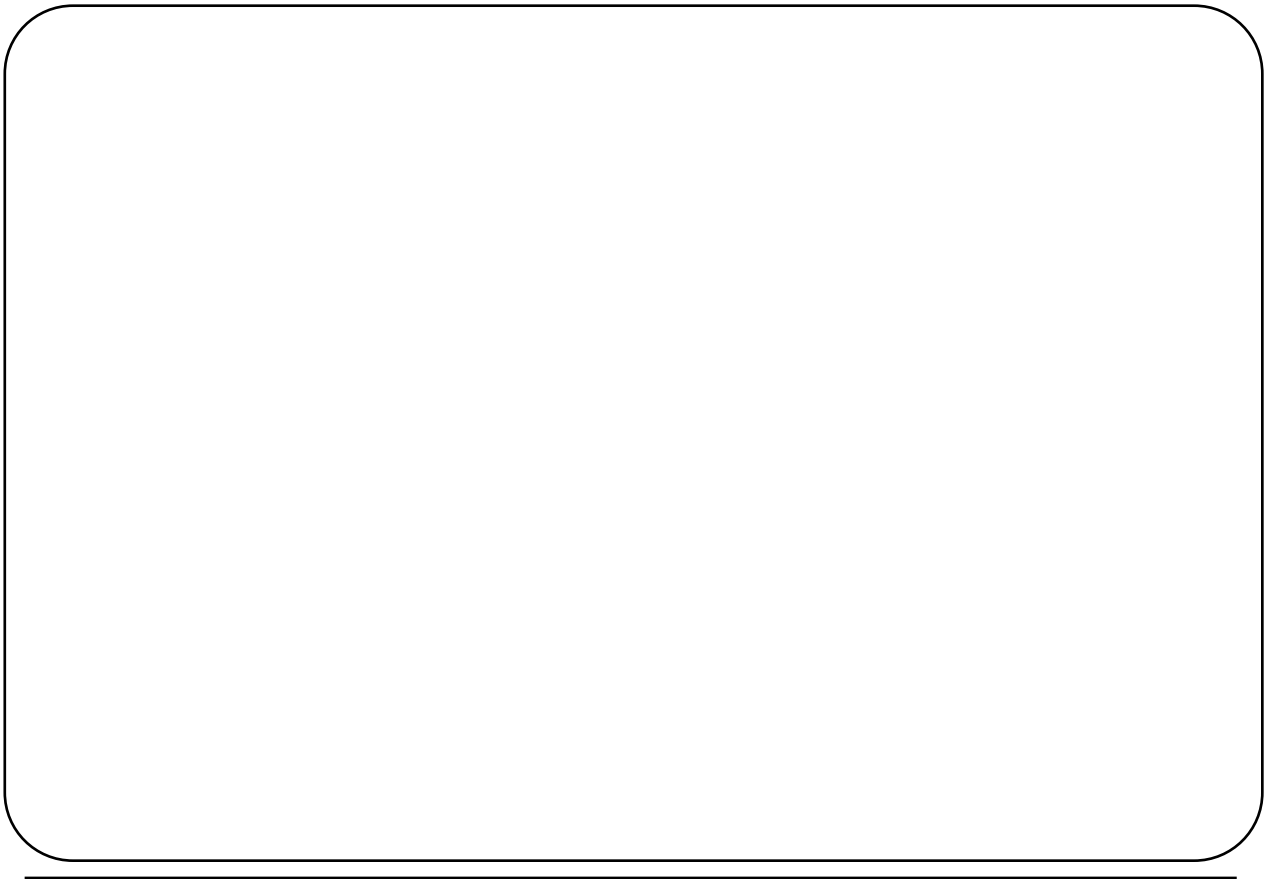
C++'s callback mechanism is extremely flexible. The mechanism can also work with inheritance hierarchies; the callback operators support polymorphism. If you declare a pointer to a Base member then it can also point to a Derived member.

The Boost libraries provide extensive support for callbacks and have encapsulated the callback mechanism in the

`boost::bind`

method. Using boost makes your code clearer and easier to maintain. See the Boost documentation for examples.





Chapter 12

12

Memory Management

- **Customising new and delete**

- globally
- class specific
- for arrays

- **Alternative forms**

- placement new
- memory allocators

- **Special forms**

- `set_new_handler`
- avoiding throwing *bad_alloc*



Chapter 12

Customising Memory Allocation

- **Overload new and delete**

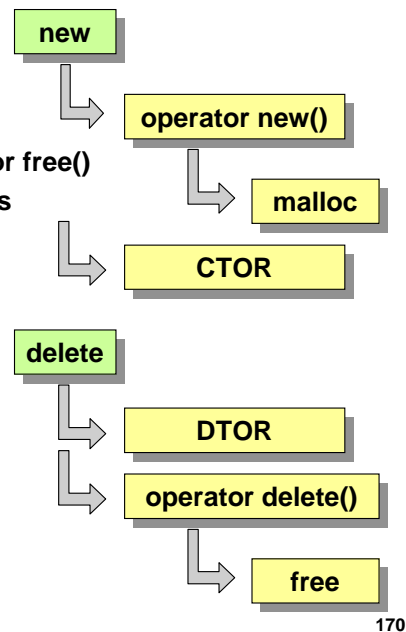
- class specific overload or
- global overload

- **Default behaviour**

- call function `operator new()` and `operator free()`
 - which call default memory allocators
- call CTOR and DTOR

- **Customisation**

- overload the operator functions



Copyright ©1994-2010 CRS Enterprises

170

C++ allows you to completely customise memory allocation by overloading two special functions

`operator new()`

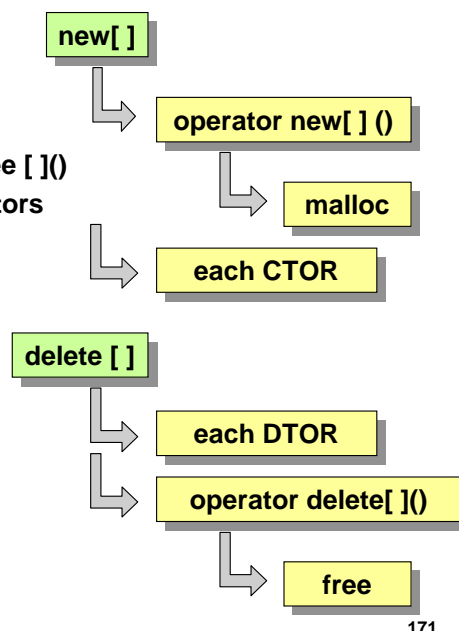
`operator delete()`

By default these functions call default memory allocators (`malloc` and `free`), but you can easily change this behaviour on a class by class basis or globally.

Note that the `new` and `delete` operators (as opposed to the overloaded `new` and `delete` functions) also call a class's constructors and destructors.

Customising Array Allocation

- **Overload new[] and delete[]**
 - class specific overload or
 - global overload
- **Default behaviour**
 - call operator new[]() and operator free[]()
 - which call default memory allocators
 - call each CTOR and DTOR
- **Customisation**
 - overload the operator functions



Copyright ©1994-2010 CRS Enterprises

171

Also you can completely customise memory allocation for arrays by overloading the two special functions

```
operator new[] ()
```

```
operator delete[] ()
```

These functions are different from the ones discussed for object overloads, but behave in a similar manner. It should be noted that memory allocation and deallocation for arrays involves a single call to these functions.

Note that the `new[]` and `delete[]` operators also call a class's constructors and destructors. However, in the case of an array, the constructors and destructors are called for each object in the array. It is a common mistake to allocate with `new[]`, but deallocate with `delete` (omitting the `[]`); this introduces a subtle bug - the destructor gets called for the first object in the array, but no other destructors get called.

Overloading Global new and delete

- Overload separately for objects and arrays
 - must use throw lists

objects

```
void* operator new (std::size_t size)
throw(std::bad_alloc)
{
    cout << "operator new()" << endl;
    return malloc(size);
}

void operator delete(void* ptr) throw()
{
    cout << "operator delete()" << endl;
    free(ptr);
}
```

arrays of objects

```
void* operator new[ ] (std::size_t size)
throw(std::bad_alloc)
{
    cout << "operator new[ ]()" << endl;
    return malloc(size);
}

void operator delete[ ](void* ptr) throw()
{
    cout << "operator delete[ ]()" << endl;
    free(ptr);
}
```

Copyright ©1994-2010 CRS Enterprises

172

You can overload the global new and delete operators as shown above. Note that the C++ standard insists on specifying a throw list as part of the signature of the globally overloaded functions

```
void* operator new (std::size_t size) throw(std::bad_alloc)
void operator delete(void* ptr) throw()
```

Similarly you can overload the global new[] and delete[] operators

```
void* operator new[ ] (std::size_t size) throw(std::bad_alloc)
void operator delete[ ](void* ptr) throw()
```

Overloading Class new and delete

- **Overload separately for objects and arrays**
 - no throw lists for class overloads

```
class Point
{
public:
    Point() { ... }
    ~Point() { ... }

    // overload new and delete for the class
    static void* operator new (size_t);
    static void* operator new[] (size_t);
    static void operator delete (void*);
    static void operator delete[] (void*);
private:
    int x;
    int y;
};
```

Copyright ©1994-2010 CRS Enterprises

173

Alternatively, you can overload the new and delete on a class by class basis. The class specific overloads are always static and must not use a throw list

```
static void* operator new (size_t);
static void operator delete (void*);
```

Similarly you can overload the class specific new[] and delete[] operators

```
static void* operator new[] (size_t);
static void operator delete[] (void*);
```

Alternative Forms of new and delete

- Alternative forms for class specific overloads

- additional parameters in overloads

```
class A
{
    static void* operator new (size_t size, int x, int y)
    {
        if(x > y) throw string("x > y");
        int* ptr = static_cast<int*>(malloc(size));
        ptr[0] = x; ptr[1] = y;
        return ptr;
    }
    static void operator delete(void* ptr, int x, int y)
    {
        // only called when an exception is thrown
        free(ptr);
    }
    static void operator delete(void* ptr)
    {
        // normal call
        free(ptr);
    }
}
```

```
pa = new (5,7) A;
delete pa;
```

Copyright ©1994-2010 CRS Enterprises

174

As if the previous customisation facilities were not enough, you can also provide an arbitrary number of further class specific overloads, each with additional parameters from the normal forms.

```
static void* operator new (size_t size, int x, int y)
```

is called when new is invoked with additional parameters as in

```
pa = new(5,7) A
```

noting the unusual syntax for new.

```
static void operator delete(void* ptr)
```

is called from the statement

```
delete pa;
```

```
static void operator delete(void* ptr, int x, int y)
```

is never called directly. This function is called only if an exception is thrown.

Using the Alternative Forms

- Alternative forms for class specific overloads
 - additional parameters in overloads

```
int main()
{
    A* pa;
    try
    {
        pa = new (5,7) A; // succeeds
        pa->Display();
        delete pa;

        pa = new (6,2) A; // fails
        pa->Display();
        delete pa;
    }
    catch(const string& message) { ... }
}

class A
{
public:
    A()
    {
        if(x > y) throw string("x > y");
    }
    static void* operator new (size_t size, int x, int y);
    static void operator delete(void* ptr, int x, int y);
    static void operator delete(void* ptr);
    ...
private:
    int x;
    int y;
}
```

Copyright ©1994-2010 CRS Enterprises

175

If an object is constructed without an exception being thrown then

```
static void operator delete(void* ptr)
```

is called when the object is deallocated. However, if an exception is thrown from within the constructor then

```
static void operator delete(void* ptr, int x, int y)
```

is called immediately.

Placement new

- **Standard memory allocators are generic**
 - inefficient
 - heap fragmentation
 - no compaction
- **Design your own memory allocator**
 - use placement new
 - to overcome these problems
 - several industrial strength allocators are available

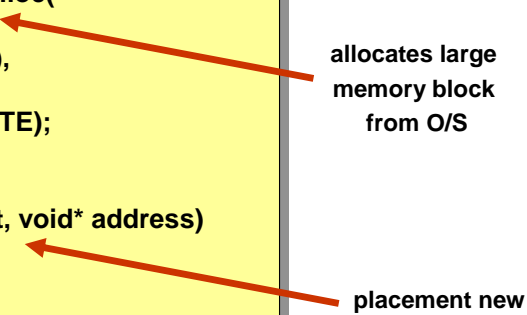
All the operator overloads discussed so far ultimately assume that a memory allocator such as malloc or free is available. However, standard memory allocators were designed for the C environment where large blocks of memory are requested from the underlying operating system. These allocators are not particularly efficient in an object oriented environment, where many small objects are continually being allocated and deallocated. These allocators also suffer from fragmentation and do not provide compaction of memory blocks.

By using C++'s placement new syntax you can design your own memory allocator to overcome the above problems. In reality, writing your own allocator is a complex task and you would be better served using one of several industrial strength allocators that are available. Nevertheless, studying placement new is quite interesting.

Placement new Overloads

- **Placement new**
 - one of the alternative forms

```
class Date
{
    static Date* AllocateMemory()
    {
        Date* array = (Date*) VirtualAlloc(
            0,
            100 * sizeof(Date),
            MEM_COMMIT,
            PAGE_READWRITE);
        return array;
    }
    static void* operator new(size_t, void* address)
    {
        return address;
    }
}
```



allocates large memory block from O/S

placement new

Copyright ©1994-2010 CRS Enterprises

177

Placement new is one of the alternative forms of the operator new function. Typically a class will provide a static method to allocate a large block of memory and then use the operator new function to customise allocation from this block. What makes placement new special is that memory has been allocated prior to the call to operator new.

Using Placement new

```
static void* operator new(size_t, void* address)
{
    return address;
}
```

to set this pointer

```
int main()
{
    // pre-allocate raw memory for objects
    Date* dates = Date::AllocateMemory();

    // place objects in raw memory
    new (&dates[0]) Date(12, 8, 1999);
    new (&dates[1]) Date(22, 5, 2000);
    new (&dates[2]) Date(31, 1, 2001);
    new (&dates[3]) Date(4, 11, 1999);
}
```

no need to save
return value

Copyright ©1994-2010 CRS Enterprises

178

Since memory has been allocated before placement new is called it makes no sense to save the return value from operator new. In fact the selected memory address is passed as one of the additional parameters to operator new since the calling program knows where the object is to be placed (in the statically allocated memory block).

Note that it is essential to return an address from operator new to set the this pointer.

A full memory allocator must expand on these ideas. Where to place objects and caching strategies are among the design decisions that need to be addressed in real allocators.

Caching Memory

- A class can be provided to cache blocks of memory
 - Blocks may be fetched using the *get* member function
 - Blocks may be added using the *put* member function

```
class cache
{
public:
    ~cache();
    void * get();
    void put(void *);
    void flush();
private:
    deque<void *> blocks;
};
```

```
void * cache::get()
{
    void * result = 0;
    if (!blocks.empty())
    {
        result = blocks.front();
        blocks.pop_front();
    }
    return result;
}
```

```
void cache::put(void * more)
{
    blocks.push_front(more);
}
```

Copyright ©1994-2010 CRS Enterprises

179

A caching strategy is often a good one to use for optimising allocation and deallocation of memory. Memory blocks of the same size can be added to or removed from the cache with very little overhead, whereas the default heap manager has the more complex task of managing memory of different sizes.

To simplify the creation of any allocation scheme it seems a wise idea to factor out the code and behaviour for a cache into a separate class, as shown above. Internally a *cache* object holds a *deque* of pointers to the cached blocks. Adding a block to the cache using *put* inserts a pointer to the block into the container, and removing from it using *get* removes an item from the container and returns it. If the cache is empty a null pointer is returned.

The cache itself holds no size information, this reduces any housekeeping overhead but it is up to the user to use cache objects in a consistent manner.

set_new_handler

```
enum { _100_MByte = 100 * 1024 * 1024 };

void low_on_memory()
{
    cout << "No more memory" << endl;
    exit(1);
}

int main()
{
    int MBytes = 0;
    set_new_handler(low_on_memory);

    while(true)
    {
        new char[_100_MByte];
        MBytes += 100;
        cout << MBytes << " MBytes allocated";
    }
}
```

- **primitive mechanism**
 - for providing recovery for failed memory allocations
- **register**
 - your own handler function

You may have noticed that C++ programmers often allocate objects dynamically, but never seem to check for memory allocation failures. If you can register a handler with the standard `set_new_handler` function you can be assured that your handler will be called in the unlikely event of a memory allocation failure.

Avoiding Throwing `bad_alloc`

- memory allocation failure
 - throws `bad_alloc` by default
 - use *nothrow* to return null pointer

```
int main()
{
    Large* p;

    p = new(nothrow) Large();
    if (!p)
        cerr << "p = 0" << endl;

    try
    {
        p = new Large();
    }
    catch(bad_alloc e) { ... }
}
```

don't throw
`bad_alloc`

normal behaviour

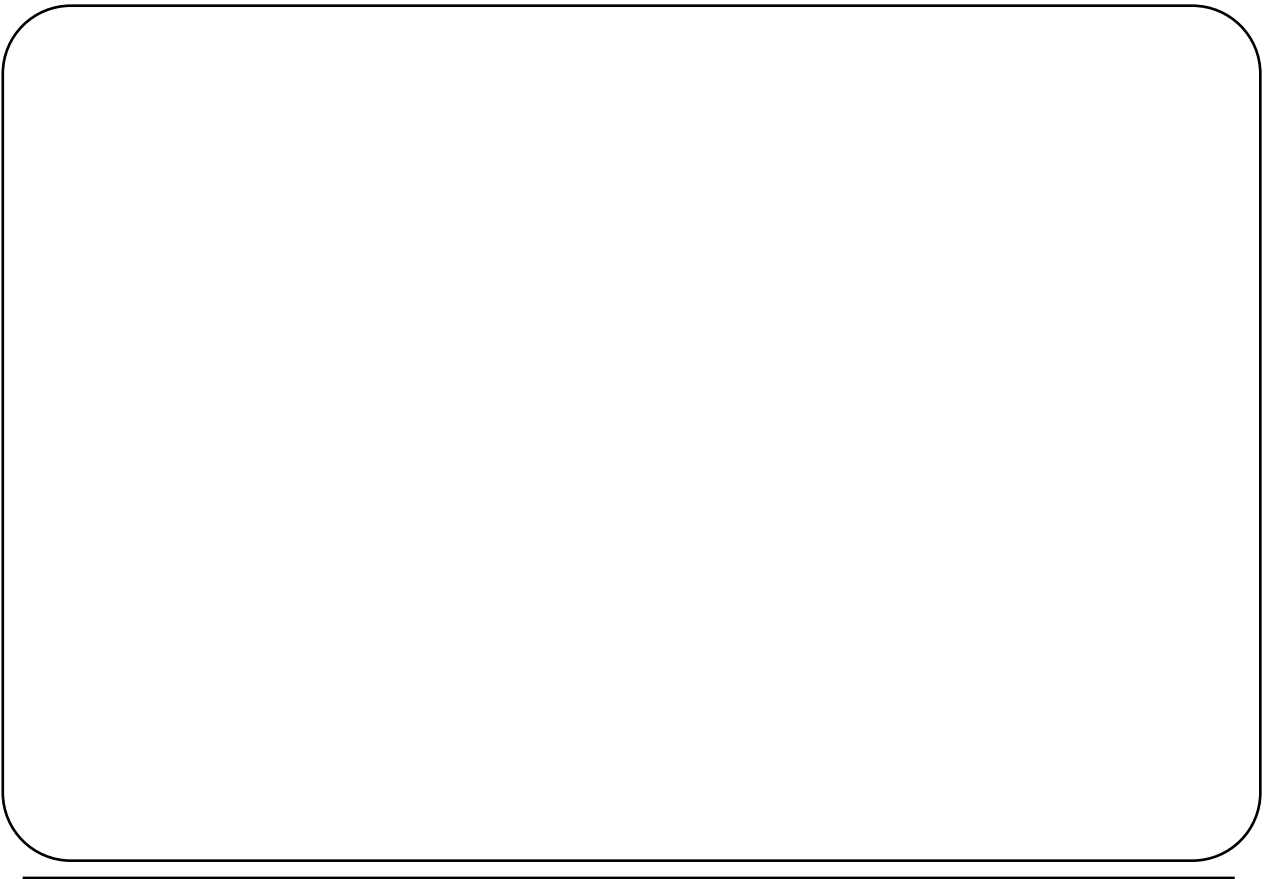
```
enum { very_big = 256 * 1024 * 1024 };

class Large
{
public:
    Large() {}
private:
    int block1[very_big];
    int block2[very_big];
};
```

Copyright ©1994-2010 CRS Enterprises

181

As a final twist, you can choose whether to throw a `bad_alloc` (default) or return a null pointer if a memory allocation fails. Throwing a `bad_alloc` is invariably a better technique, but returning a null pointer might be acceptable if you need compatibility with legacy code.



Chapter 13

13

Singleton

- **Basic Idiom**
 - Implementing the Singleton
 - Instantiating the Singleton
- **Don't use Static Functions**
 - too difficult to control
- **Myers Singleton**
 - interesting design
- **Double Checked Locking**
 - anti-pattern



Chapter 13

Basic Idiom

- **Implement as a class**

- control creation
 - public instance method
 - protected constructor
- restrict copying
 - protected copy constructor
- restrict assignment
 - return pointer to singleton

```
class Singleton
{
private:
    static Singleton* pInstance;
protected:
    Singleton(const Singleton&) {}
    Singleton() {}
    ~Singleton() {}
public:
    static Singleton *instance();
};
```

Singletons are notoriously difficult to implement; the usual method is shown above. A Singleton class is defined which controls creation of a single object, and restricts how users copy and assign this object.

There must only be ONE!

Instantiation

- **Create instance on heap**
 - use static pointer
 - **pInstance**
- **Provide method to retrieve instance**
 - create instance on first request

```
class Singleton
{
private:
    static Singleton* pInstance;
protected:
    Singleton(const Singleton&) {}
    Singleton() {}
    ~Singleton() {}
public:
    static Singleton *instance();
};
```

```
Singleton* Singleton::instance()
{
    if (pInstance == 0) pInstance = new Singleton;
    return pInstance;
}

Singleton* Singleton::pInstance = 0;
```

Copyright ©1994-2010 CRS Enterprises

186

The Singleton class creates its one and only instance on the heap and keeps track of the instance using a static pointer

pInstance

The class provides a dedicated method to retrieve this instance

Singleton* Singleton::instance()

This method creates the instance on the first request. Subsequent requests return the same instance pointer.

Don't use Static Functions

- **static functions can't be virtual**
 - can't use polymorphism in derived classes
- **initialization and cleanup difficult**
 - definitions not in one place
 - static defined outside class
- **can't cope with dependencies**
 - such as default font depending on the printer port

```
class Font {};  
class PrinterPort {};  
  
class MyOnlyPrinter  
{  
private:  
    // All data static  
    static Font defaultFont;  
    static PrinterPort printerPort;  
public:  
    // all methods static  
    static void AddPrintJob() { ... }  
    static void CancelPrintJob() { ... }  
};
```

It might seem easier to define singletons in terms of static functions, but we strongly advise against it.

Since static functions can't be virtual, we would not be able to use polymorphism in derived classes. Furthermore, initialization and cleanup becomes difficult because definitions are not in one place. Finally, using statics introduces uncertainties in the order of initialization and this means we can't cope with dependencies such as a default font depending on a printer port.

Myers Singleton

- **Uses static objects**
 - initialized on creation
 - unlike static variables
- **Cleanup automatic**
 - compiler calls DTOR through *atexit*

```
class Singleton
{
protected:
    Singleton(const Singleton&) { }
    Singleton() { }
    ~Singleton() { }
public:
    static Singleton* instance();
};
```

```
Singleton* Singleton::instance()
{
    // static objects are initialized when control first
    // passes through this function
    // not to be confused with static variables which are
    // initialized at program load time

    static Singleton instance;
    return &instance;

    // compiler ensures DTOR is registered as an atexit function
}
```

Copyright ©1994-2010 CRS Enterprises

188

Myers has produced an interesting variant of the singleton design. Superficially, the Myers singleton looks like the standard design. However, Myers exploits an unusual feature of C++; the use of static objects as local variables.

When a local variable is declared as static, its behaviour differs depending on whether it is a built in type or an object of a user defined class. If the variable is a built in type it will be initialized at program load time along with all other statics. However, if the variable belongs to a user defined class, as in our case, then it is initialized when control passes through its containing function. Furthermore, the compiler ensures its destructor is registered with the standard library `atexit()` function to provide deterministic clean up at the end of the program.

Double Checked Locking Anti-Pattern

- **Double Checked Locking Pattern**

- introduced for single CPU systems
- provides performance gain
- doesn't work with modern processors

```
Singleton* Singleton::instance()
{
    if(!pInstance) // first check
    {
        Lock locker; // CTOR locks
        if(!pInstance) // second check
        {
            pInstance = new Singleton;
        }
    } // DTOR unlocks
    return pInstance;
}
```

Copyright ©1994-2010 CRS Enterprises

```
class Lock
{
public:
    Lock()
    {
        EnterCriticalSection(&cs);
    }
    ~Lock()
    {
        LeaveCriticalSection(&cs);
    }
};
```

189

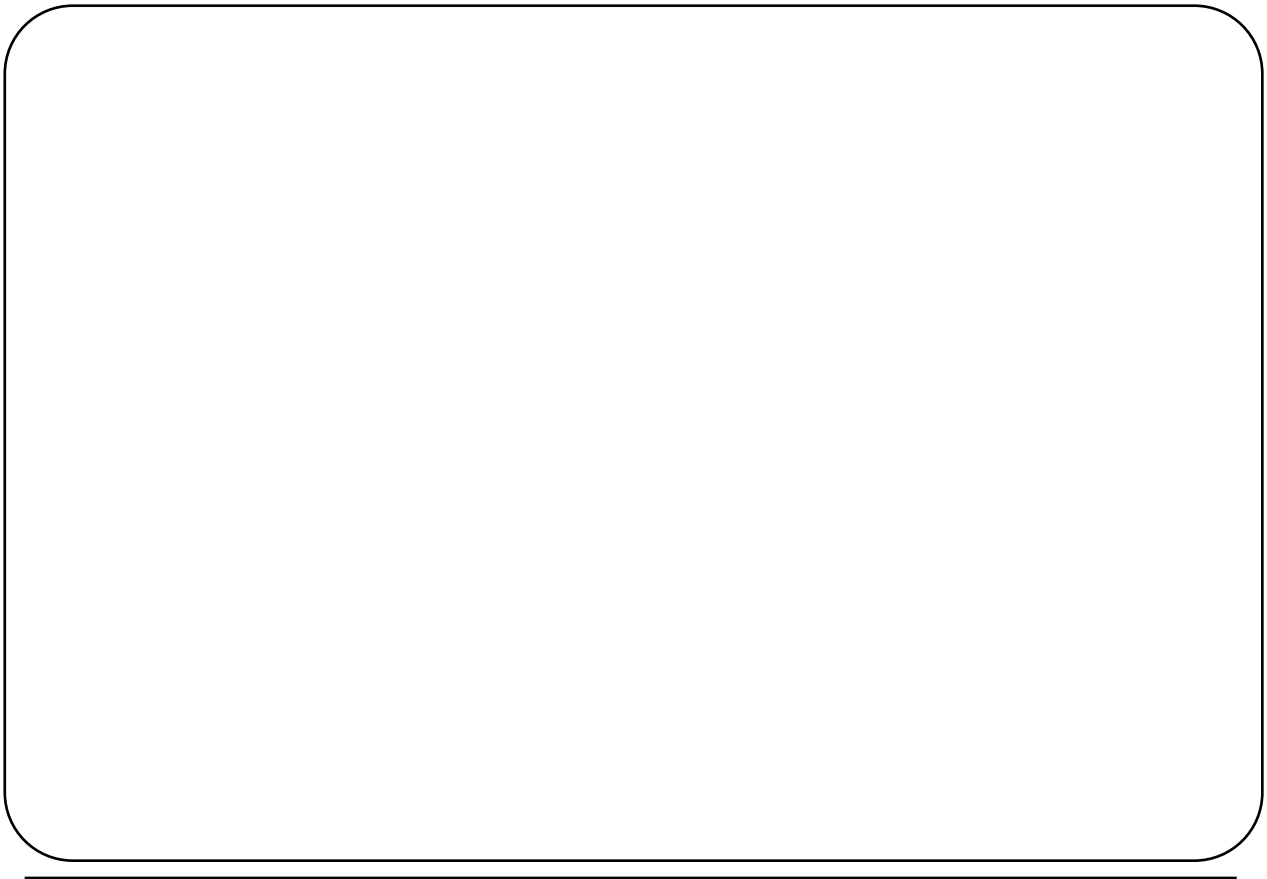
The double check locking pattern has been in use for some time on single processor CPU systems. This pattern is designed to optimise locking objects in a multi threaded environment. Unfortunately, recent advances in instruction pipelining and caching CPU memory on both single and multi-processor systems have invalidated this pattern. It is now a recognised anti-pattern. Nevertheless, it is an interesting design

The example above shows the anti-pattern used to implement the instance method of the Singleton class. The method relies on the Lock class to synchronize between different threads; the Lock class uses the

resource acquisition is initialization

idiom to call the underlying Windows Critical Section locking mechanism (similar mechanisms are available in Unix).

Note that the anti-pattern first checks to see if the singleton exists without synchronization. This optimization succeeds if the singleton already exists. If the singleton has not been created, the pattern tries again, but this time using synchronized access to avoid a race condition with another thread (this is the part that doesn't work any more).



Chapter 14

14

Copyright ©1994-2010 CRS Enterprises

191

Object Factories

- **Virtual Constructors using Switches**
 - why they are poor design
- **Virtual Construction using Cloning**
 - better design
 - objects must exist prior to cloning
- **Exemplar Method for Object Factories**
 - excellent design pattern



Chapter 14

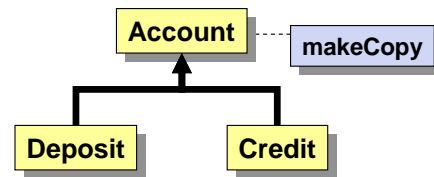
Virtual Constructor

- **Virtual constructors**

- allow objects to be created without the client having to specify a concrete type
- no C++ language support
- but many idioms implement the concept

- **A major goal**

- allow programs to be constructed based on interfaces
- independent of concrete types



```

int main()
{
    Account* bruce = CreateAccount("Deposit");
    Account* dave = CreateAccount("Credit");
    takeMoney(bruce);
    takeMoney(dave);

    Account* bruce2 = bruce->makeCopy();
    takeMoney(bruce2);

    Account* dave2 = dave->makeCopy();
    takeMoney(dave2);
}
  
```

Copyright ©1994-2010 CRS Enterprises

193

Virtual constructors allow objects to be created without the client having to specify or indeed be aware of the concrete type of the object. This allows the client to work with interfaces and not be concerned with implementation issues. Unfortunately, there is no C++ language support for virtual constructors, but many idioms have been put forward that implement the concept. The major goal is to allow programs to be constructed that are independent of concrete types.

Consider the above example. We wish to create different types of accounts for bruce and dave, but don't want to have to explicitly refer to the implementation class. What we want is for CreateAccount to create the appropriate account behind the scenes and simply return an interface pointer. That way, if a new account type is added or an existing type is changed, our program is decoupled and isolated from the changes. Similar considerations apply if bruce and dave want to make copies of their accounts.; they just use makeCopy as defined in the Account interface.

Switch Statement

- **Poor design**

- program is not entirely free of knowledge about specific sub-classes
- virtual constructor is not easy to maintain
 - switch statements almost always bad

```
Account* CreateAccount(string accountType)
{
    // Virtual CTOR using a switch statement
    // poor design

    Account *pAccount = 0;
    if (accountType == "Deposit")
        pAccount = new Deposit();
    if (accountType == "Credit")
        pAccount = new Credit();

    return pAccount;
}
```

```
Account* bruce = CreateAccount("Deposit");
Account* dave = CreateAccount("Credit");
```

Copyright ©1994-2010 CRS Enterprises

194

In our first attempt at a virtual constructor, we can use a switch statement to work out which type of account is required. However, this means the calling program is not entirely free of knowledge about specific sub-classes being used. Even if we can accept this dependency, this is a very poor design; switch statements are notoriously difficult to maintain. In a large application, switch statements like the above will become distributed throughout the system. Then, when a new account is added, deleted or changed, we must edit every switch statement in the application - an odorous task.

Making Copies with Switch

- Use `dynamic_cast` to check object type
 - still messy, but safe

```
Account* bruce2 = bruce->makeCopy();  
Account* dave2 = dave->makeCopy();
```

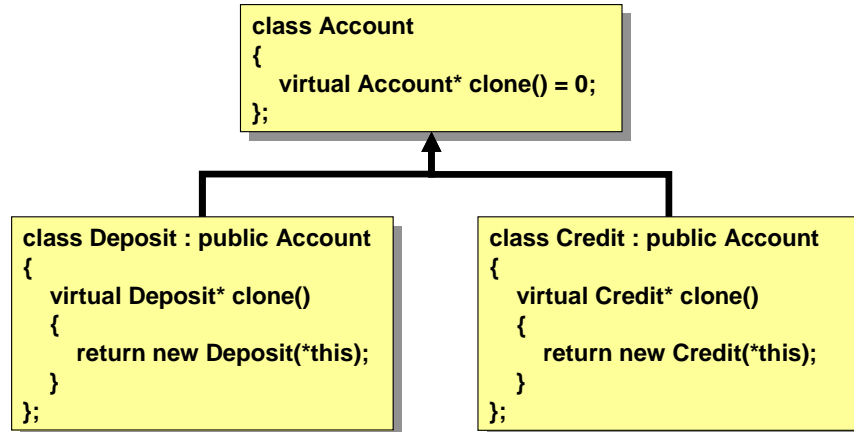
```
Account* Account::makeCopy()  
{  
    Account *copy;  
    if (dynamic_cast<Deposit*>(this))  
        copy = new Deposit();  
    else  
        copy = new Credit();  
  
    return copy;  
}
```

If we continue with our design, based on switch statements, we can hide implementation details from the client when we make a copy of our objects. The `dynamic_cast` can check the type of the object and then create a copy of appropriate type. The design is still messy, but safe.

Cloning

- Copying can be effected
 - using a polymorphic clone method
 - avoids the switch statement
 - client still decoupled from implementation

```
Account* bruce2 = bruce->clone();
Account* dave2 = dave->clone();
```



Copyright ©1994-2010 CRS Enterprises

196

To improve our design, we must attempt to avoid using switch statements. Switch statements are regarded as the antithesis of good design.

Standard techniques using polymorphism lead to a much cleaner design. With polymorphism, the switch is gone and a clone method will create a copy of our objects for us. Note that as far as we are concerned, according to the `Account` interface, the clone method returns an `Account*`, thereby decoupling us from the implementation.

```
virtual Account* clone()
```

However, as you can see from the implementation code, the `Deposit` and `Credit` classes return concrete objects

```
virtual Deposit* clone()
```

```
virtual Credit* clone()
```

Technically, we say clone has a covariant return type.

Exemplars

- **Removing the switch from object creation is more difficult**
 - use the Exemplar idiom
- **Create exemplars**
 - in each concrete class
 - using a build method
 - register build method with base class
- **Client asks base class to create object**
 - base determines which type of object
 - calls the appropriate build method
 - to create the real object

Copyright ©1994-2010 CRS Enterprises

197

Removing the switch statement from object creation is much more difficult. An excellent way to do this is to make use of the Exemplar idiom.

With this idiom each concrete class is responsible for providing an exemplar object or, more simply, a build method that creates an object of the class. Each concrete class has to register this build method with the base class (this must occur before any objects are created - how?), so that the base class ends up with a collection of build methods, one for each concrete class.

When the client requests the base class to create an object, the base class searches this collection and calls the appropriate build method.

Exemplar - Client Code

- **Client is decoupled from implementation classes**
 - use a static *make* method for creating objects

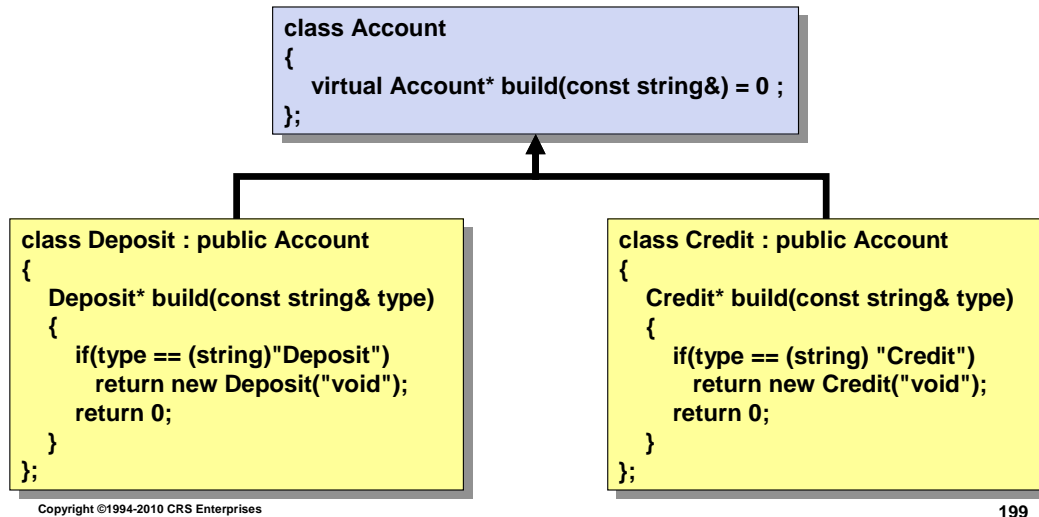
```
int main()
{
    Account* bruce = Account::make("Deposit");
    takeMoney(bruce);

    Account* dave = Account::make("Credit");
    takeMoney(dave);
}
```

The client code uses a static make method from the base class, Account, to create an object. This is the virtual constructor.

Exemplar - build method

- **build method**
 - constructs an *exemplar* which can be tailored later
 - e.g. change account name from "void"



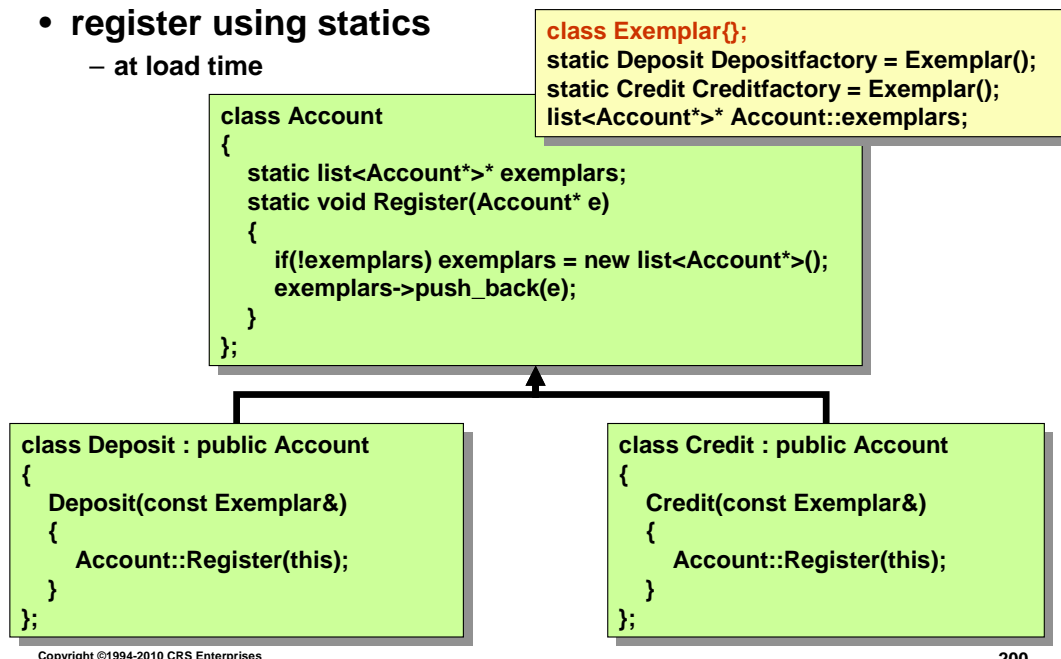
Each class defines its own build method which will be registered with the base class (see next slide). Later, the base class will call the build method with the string passed in by the client. The build method must check this string for a match and then create the required object. If the string doesn't match, it means that the client doesn't want one of these objects and the class must return a 0 pointer to indicate rejection.

Once an object has been created, attributes can be added or modified, depending on what is defined in the Account interface. For example, the Account name could be changed later from void to a more suitable name.

Exemplar - registering build

- register using statics

- at load time



Each derived class is responsible for registering itself with the base class and the base class maintains a collection of registrations. The derived class has the tricky problem of registering the exemplar before any derived objects are created; the solution is to use statics.

Each derived class creates a static instance of an Exemplar object. Note that the Exemplar class is an empty class whose only purpose is to trigger a call to the constructors

```
Deposit(const Exemplar&)
```

```
Credit(const Exemplar&)
```

and because we are dealing with statics, this call is made at program load time, before any derived objects are created. We need to be careful of the order in which static code is initialized; this depends on the order in which modules are compiled. The best solution is for the base class to create a pointer to a list of exemplars, rather than deal with the list directly using static initialization. If we were to use the list directly (rather than a pointer to the list), the exemplars might get created before the base class's list is initialized and the system would fail. By using a pointer we can check if the list has already been created, if not we can create it there and then.

Exemplar - Virtual Construction

- **Virtual Constructor**

- delegates to build methods
- searches collection until non zero pointer returned
 - more efficient algorithms are possible

```
static Account* Account::make(const string& type)
{
    Account *p = 0;
    list<Account*>::iterator i;
    for(i = exemplars->begin(); i != exemplars->end(); ++i)
    {
        p = (*i)->build(type);
        if(p)break;
    }
    return p;
}
```

Copyright ©1994-2010 CRS Enterprises

201

The last piece of the jigsaw is the virtual constructor itself

```
static Account* Account::make(const string& type)
```

The virtual constructor searches the list of exemplars, calling the build method for each class in turn. The virtual constructor forwards an identifying string to the build method. As discussed previously, if a class rejects this string it will return a 0 pointer. The search goes on until either the list is exhausted (the construction fails) or one of the classes accepts the string, builds the object, and returns a valid pointer.

Copyright ©1994-2010 CRS Enterprises

Chapter 15

15

Copyright ©1994-2010 CRS Enterprises

203

Multiple Inheritance

- Problems with Multiple Inheritance
- Virtual Inheritance
- Mix-In Classes
- Interface Classes



Chapter 15

Multiple Inheritance

- **CompanyCar combines the interfaces of TaxableItem, Vehicle and its own additional features**

```
class TaxableItem
{
public:
    Money Get_taxable_value();
};
```

```
class Vehicle
{
public:
    int Get_engine_capacity();
};
```

```
class CompanyCar : public TaxableItem, public Vehicle
{
public:
    Date Get_renewal_date();
};
```

- **There are problems with multiple inheritance ...**
 - what if there are name collisions in the parent classes?
 - what if the parent classes have a common base class?

Copyright ©1994-2010 CRS Enterprises

205

Multiple inheritance allows a derived class to inherit from more than one direct base class. For example, the CompanyCar class shown above inherits the capabilities of two base classes: TaxableItem and Vehicle.

An instance of CompanyCar can determine its renewal date using its Get_renewal_date() function. It can also call TaxableItem::Get_taxable_value() to calculate the cost of the company car in terms of deductions from the tax-free allowance. Similarly, the Vehicle::Get_engine_capacity() function can be called to determine the size of the car's engine.

Note: if you want each of the base classes to be a public base class, you must explicitly specify the public keyword in front of each base-class name. If you forget one of the public keywords, the default mode of inheritance is private inheritance.

Note: there are several technical difficulties associated with the use of multiple inheritance, as pointed out in the slide above. For these reasons, many companies have a policy of avoiding multiple inheritance if at all possible.

Multiple Inheritance

- **Derived class may have more than one base class**
 - *multiple inheritance*
- **Only use if the concept is meaningful**
 - easily abused and can lead to confusing code

```
class control_switch
{
public:
    void set();
    void reset();
};
```

```
class clock
{
public:
    time now() const;
};
```

```
class timer_switch
: public control_switch,
  public clock
{
public:
    void set_on_time(const time &);
    void set_off_time(const time &);
};
```

Copyright ©1994-2010 CRS Enterprises

206

C++ allows a class to have any number of direct base classes allowing a class to combine the behaviour of many base classes.

Multiple inheritance is quite a controversial subject. Some commentators compare the use of multiple inheritance with the use of goto in structured programming. This is perhaps an over-statement; it is the misuse of multiple inheritance that leads to confusion.

Used carefully and appropriately, multiple inheritance is an elegant technique that fits the bill in many design situations; for instance, CORBA programming depends on multiple inheritance to allow objects to support multiple interfaces.

The example in the slide illustrates the basic syntax for multiple inheritance. The timer_switch class inherits from control_switch and clock:

```
class timer_switch : public control_switch, public clock { ... };
```

Notice that the public keyword must be used before each base-class name if you want to use public inheritance for each base class. If you omit the public keyword, the default is private inheritance.

Multiple Inheritance Scoping Issues

- By default there is no overloading across different base class scopes
 - Identically named functions in different base classes must be resolved either within the class or at the point of call

```
class control_switch
{
public:
    void set();
};
```

```
class clock
{
public:
    void set(const time &);
};
```

```
class timer_switch
: public control_switch,
  public clock
{
public:
    using control_switch::set;
    using clock::set;
};
```

Copyright ©1994-2010 CRS Enterprises

207

Identifiers in multiple inheritance are resolved the same way as in single inheritance. Local scope is checked first, followed by the class scope for the derived class. If the identifier has still not been located, the search continues up the class hierarchy branches, and finally to global scope. Base classes along different branches in an inheritance hierarchy are given the same scope, even if they are at different levels in the hierarchy.

If the same function name is found in two different paths, a scope ambiguity error occurs when you try to call the function. Consider the `control_switch` and `clock` classes shown in the slide:

```
timer_switch heater_control;
heater_control.set(); // ambiguity
heater_control.control_switch::set(); // resolves the ambiguity
```

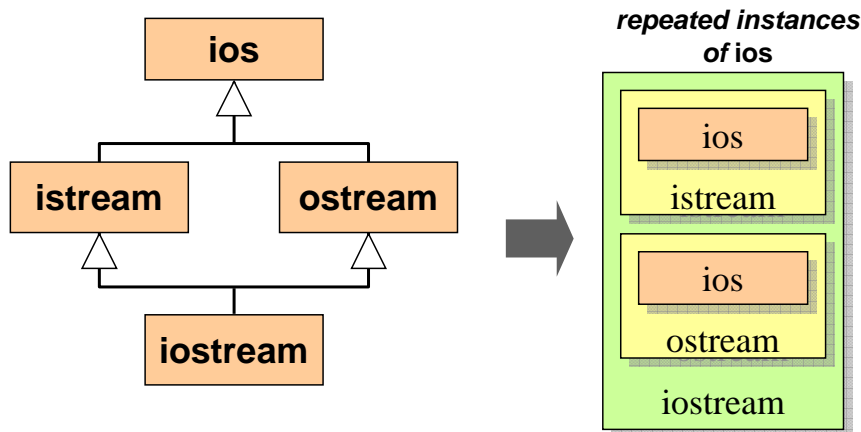
The first call is ambiguous; the compiler cannot decide whether to call `control_switch::set` or `clock::set`, despite the fact that the two functions have different signatures. The second call disambiguates matters by specifying the `control_switch::set` function explicitly. A better approach is to use a `using` declaration in `timer_switch`:

```
class timer_switch : public control_switch, public clock
{
    using control_switch::set;
    using clock::set;
};
```

This brings the `control_switch::set` and `clock::set` functions into the scope of `timer_switch`, so that they behave like overloaded functions in the same class.

Repeated Inheritance

- A class may be inherited more than once
 - via different derivation paths
 - normally results in duplicate sub-objects for the base class



Copyright ©1994-2010 CRS Enterprises

208

Multiple inheritance opens up the possibility of repeated inheritance, where a base class is inherited more than once in an inheritance graph. One example of this is to be found in the C++ library (the version described here is simplified, and corresponds to what is sometimes known as Classic I/O Streams): an `istream` is for stream input and an `ostream` is for stream output; both these classes inherit their basic buffering features from the `ios` class; the `iostream` class supports both stream input and output, and therefore is derived from both `istream` and `ostream`.

The question is, in this case, what does it mean for `iostream` to have inherited from `ios` twice? By default the compiler simply assumes that there are two instances of `ios` in an `iostream` – one in `istream` and one in `ostream` – and this means that in effect there are two buffers operating on two different underlying streams!

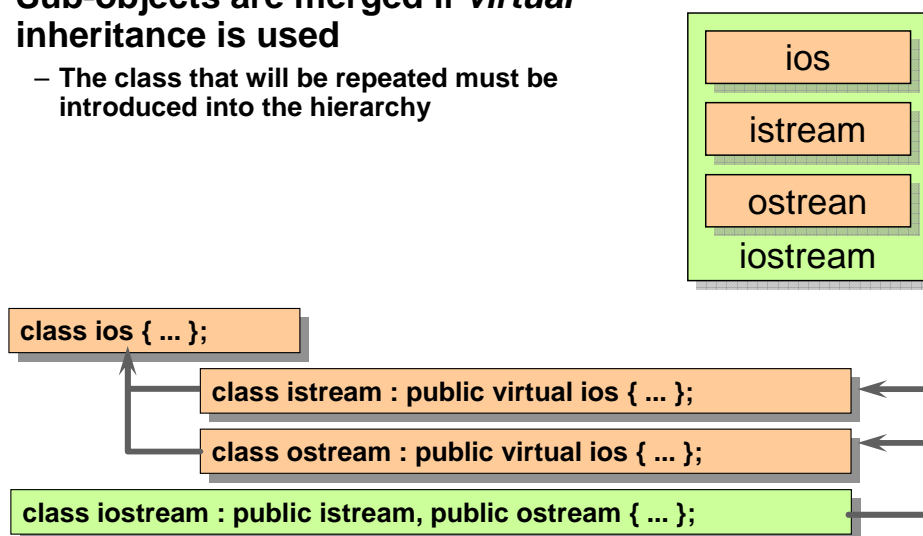
A more reasonable interpretation in this case would be that `iostream` has simply inherited the capabilities offered in `ios`, and that repeatedly inheriting from `ios` will have no further effect, i.e. the instances are merged together and there is only one. The way to express this in C++ is to use virtual inheritance: the class to be merged must always be introduced as virtual in the list of base classes.

Accommodating repeated inheritance in the manner described requires planning and anticipation that a class will be used in such hierarchies.

virtual Base Classes

- Sub-objects are merged if *virtual inheritance* is used

- The class that will be repeated must be introduced into the hierarchy



Copyright ©1994-2010 CRS Enterprises

209

For every occurrence of a common base class declared as virtual, there is only one copy in the most derived class. If a class appears several times in the hierarchy there will be a separate instance for each non-virtual occurrence, and one common instance shared between all the virtual occurrences.

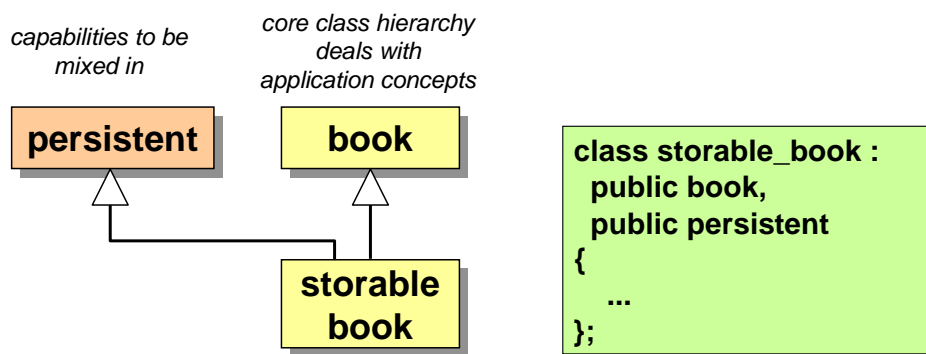
One of the problems with this technique is that the virtual keyword must be attached to the base classes. If these parents have been used elsewhere in another case of multiple inheritance, where a common ancestor is specifically not intended to be virtual, then there is a clash of interests. Such a clash can really only be resolved by duplicating the parent class with two versions, one with a normal parent and one with a virtual parent.

When a base-class constructor takes arguments, it is usually the job of the immediate derived class to pass these parameters to the base-class constructor. However, we have a problem if the base class is virtual and appears more than once in a multiple inheritance hierarchy: which of the derived classes should be responsible for initialising the virtual base class?

This problem is resolved by suppressing the construction of virtual base class by the intermediate derived classes. Instead, the most derived class takes on the responsibility of initialising the virtual base class. Furthermore, virtual base classes are guaranteed to be constructed first, regardless of their order in the hierarchy.

Mix-in Classes

- A *mix-in* is used for adding capabilities to other classes
 - each class defines a capability that derived classes can *mix-in*
 - mix-in classes are normally totally unrelated
 - avoiding many of the problems normally associated with multiple inheritance



Copyright ©1994-2010 CRS Enterprises

210

The problem of repeated inheritance, discussed on the previous page, most often occurs when a class inherits from two similar base classes. `iostream` inherits from `istream` and `ostream`, which are similar in purpose and implementation and inherit from the same ‘grandparent’ class, `ios`. To overcome the problem of repeated inheritance, `istream` and `ostream` have to declare `ios` as a virtual parent.

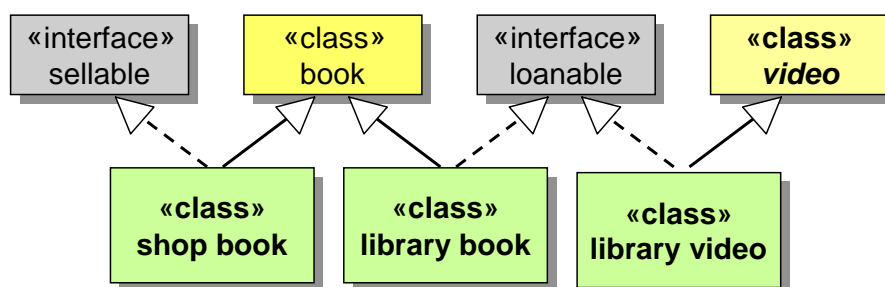
These difficulties do not arise if a class inherits from two completely orthogonal classes. For instance, the `storable_book` class in the slide inherits from `book` (for its book-related operations) and from `persistent` (for its operations to read and write data to disk). `book` and `persistent` are two completely different classes, and we are therefore extremely unlikely to suffer from repeated inheritance.

Furthermore, the risk of name clashes is reduced as well, because the function names in `book` are likely to be quite different from the function names in `persistent`. If we do happen to experience a name clash, `storable_book` can use a `using` declaration to republish the appropriate functions as necessary.

Designs that use mix-in classes in this way are very flexible, and allow new classes to be introduced with remarkably little effort. For example, if we define new classes such as `video` and `cassette`, we can define `storable_video` and `storable_cassette` classes immediately, simply by using multiple inheritance to combine the behaviour of `storable` with that of `video` and `cassette` respectively.

Interface Classes

- It is sometimes desirable to provide many alternative implementations to the same class interface
 - Different implementations should be used uniformly by the client
- What does multiple interface inheritance mean?
 - Interface as role: multiple roles
 - Interface as capability: support for many capabilities
 - Interface as service: provider of multiple services



Copyright ©1994-2010 CRS Enterprises

211

In the slide above, sellable and loanable are interfaces. They define the set of operations that all sellable and loanable objects should support, without regard for the type of item being sold or lent.

sellable and loanable do not provide any behaviour themselves; all they do is specify a common set of operations that can be implemented by derived classes. In contrast, the book and video classes are concrete classes, and provide the implementation and representation details for a book and a video.

Equipped with the sellable and loanable interfaces, and the book and video classes, multiple inheritance can be used to combine the interfaces and concrete classes to produce shop_book, shop_video, library_book, and library_video as follows:

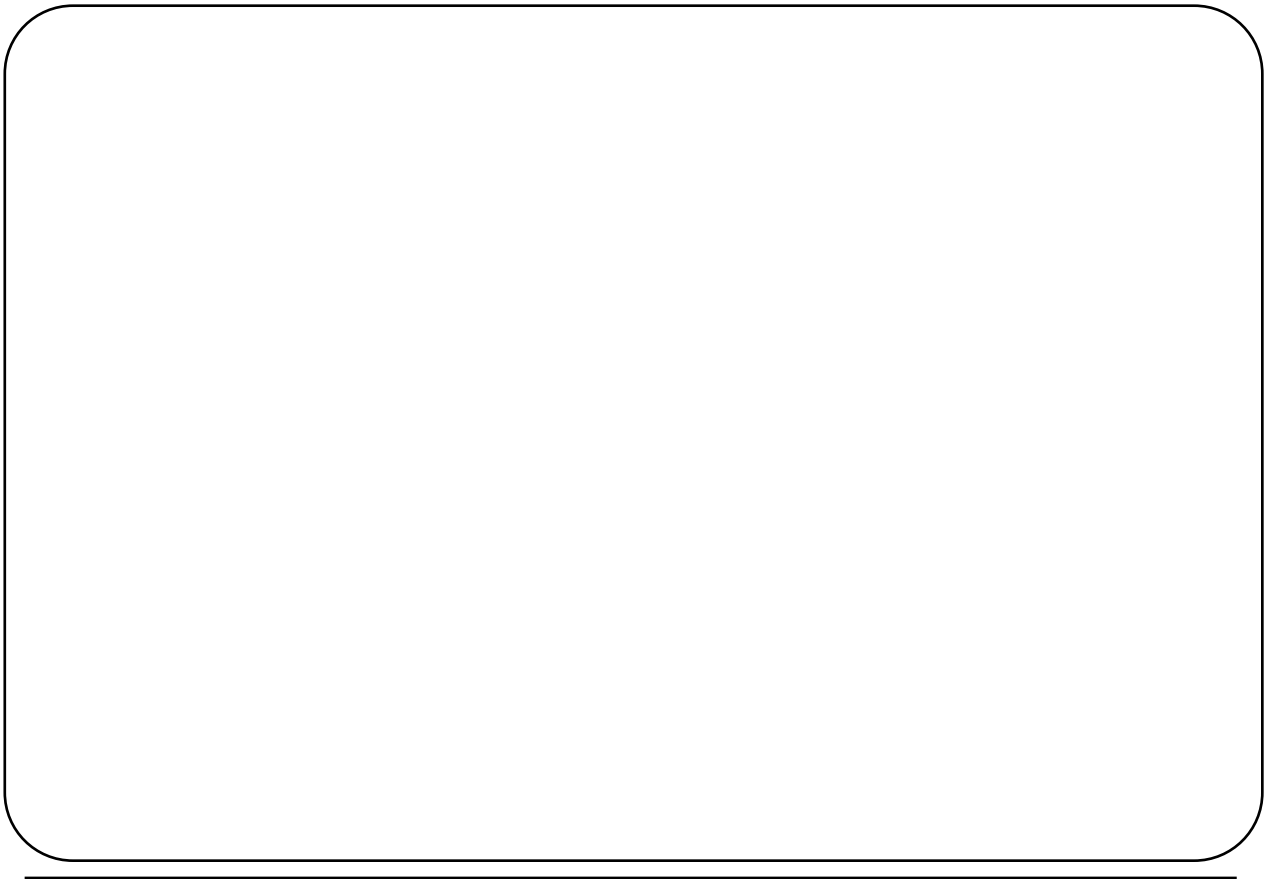
```

class shop_book : public book, public sellable { ... };
class shop_video : public video, public sellable { ... };
class library_book : public book, public loanable { ... };
class library_video : public video, public loanable { ... };
  
```

This use of interfaces ensure consistency across the system. For example, any object that is sellable will have the same protocol, as defined by the sellable interface. Interfaces can also be used in combination, so that a class can support any number of interfaces:

```

class loanable_sellable_book :
    public book, public loanable, public sellable { ... };
  
```



Chapter 16

16

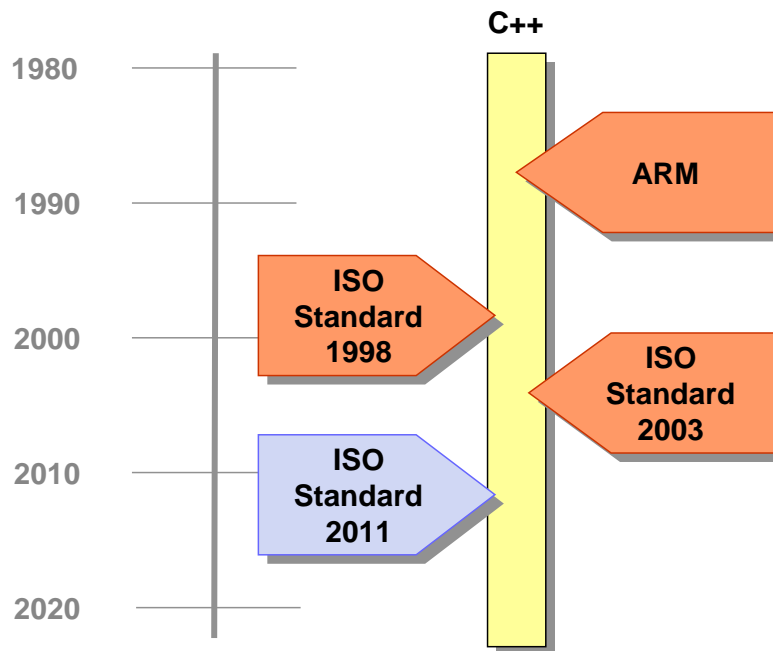
C++ 2011

- **Language Changes**
 - lots of them
- **Library Changes**



Chapter 16

C++ Standards



Copyright ©1994-2010 CRS Enterprises

215

C originated in the years 1969-1973, in parallel with the early development of the Unix operating system. C was written by Dennis Ritchie and Brian Kernighan and underwent many changes in its early years..

C++ was designed for the UNIX system environment by Bjarne Stroustrup as an object oriented version of C. C++ was first standardized in 1998. Before that Bjorn Stroustrup and Mary Ellis's Annotated Reference Manual (ARM) was effectively the standard. The standard had minor additions in 2003, but it was not until 2011 that a major revision has taken place. At the time of writing, the 2011 standard still needs ratification (August 2011?).

Core Language Enhancements ...

- **Rvalue references and move constructors**
 - optimisation with deep copies
- **Constant expressions**
 - compile time definitions for optimizations
- **More constness syntax for initialization**
 - of 'plain old data'
- **Extern templates**
 - inhibit generation of templates in a compilation unit

... Core Language Enhancements

- **Many other language enhancements ...**
 - type inference
 - range based loop
 - lambda functions
 - alternative declarations of return types
 - constructors calling other constructors (as in Java)
 - type safe enums
 - **>>** can be template terminator
 - **explicit** now applies to operator casts (as well as CTOR casts)
 - template aliases
 - variadic templates
 - new Unicode strings

... Core Language Enhancements

- **And still more ...**
- **Threading is now part of the language**
 - memory model defined
 - implementations are library based
 - e.g. Boost Threading
- **Upgrades to the STL**
 - tuples
 - hash tables
 - RegEx
 - Smart Pointers
 - Random Numbers
 - Metaprogramming
 - Type traits

R-Value Refs (Move Semantics) ...

- **Move problem**

- function calls create temporaries
- inefficient if deep copies are involved because
- copies have to be destroyed when we could just swap pointers

```
X func()
{
    // X is a complex type with deep copying semantics
    X temp;
    // do something with temp
    return temp
}
```

```
X x;
x = func();
```

temp must be destroyed

x is a deep copy of temp

why not just swap pointers
to avoid deep copying

... R-Value Refs (Move Semantics)

- Use Rvalue references on functions

– X&& x

```
void foo(X& x);    // lvalue reference overload void
void foo(X&& x);   // rvalue reference overload
```

- and on copy CTOR and assignment

```
X(const X& original);
X(X&& original)

X& X::operator=(const X& rhs);
X& X::operator=(X&& rhs);
```

Rvalue Refs (Perfect Forwarding) ...

- Forwarding problem

- factory creates deep copy before forwarding to client

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg arg)
{
    return shared_ptr<T>(new T(arg));
}
```

- if we pass by reference arg can't be an r-value

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg& arg)
{
    return shared_ptr<T>(new T(arg));
}
```

```
factory<X>(func()); // error if func returns by value
factory<X>(41); // error
```

... Rvalue Refs (Perfect Forwarding)

- Use Rvalue references
 - Arg&& arg

```
template<typename T, typename Arg>  
shared_ptr<T> factory(Arg&& arg)  
{  
    return shared_ptr<T>(new T(std::forward<Arg>(arg)));  
}
```

Type Inference

- **decltype**

- copies type

```
int x;  
  
// define a variable with the same type as x  
decltype(x) y;
```

- **auto**

- infers the type

```
// int (*fp)(int, int) = &f;  
auto fp = &f;  
result = fp(10, 20);
```

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

Lambda Functions ...

- **creates a function inline**
 - can be used wherever an expression can be used

```
auto fp =  
    [](int x, int y) -> int  
    {  
        int temp = x + y;  
        return temp;  
    };
```

```
int result = fp(10, 20);
```

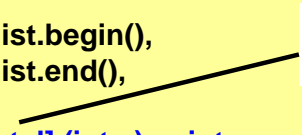

... Lambda Functions

- lambdas support closures

```
vector<int> mylist = { 2, 3, 5, 7, 11, 13, 17, 19 };
int total = 0;

for_each( mylist.begin(),
          mylist.end(),
          [&total] (int x) -> int
          {
              total += x;
              return total;
          }
        );
```

closure on
reference to total

A black arrow points from the text box 'closure on reference to total' to the [&total] part of the lambda function definition in the code.

Type Safe enums

```
enum class E : unsigned long { E1 = 1, E2 = 2, Ebig = 0xFFFFFFFF0U };  
  
void f(enum class E x)  
{  
    // conversion to int not allowed  
    cout << x << endl;  
  
    // conversion from int not allowed  
    x = 1;  
}  
  
int main()  
{  
    f(E::E1);  
    f(E::E2);  
    f(E::Ebig);  
}
```

define underlying integral type

old C++ enum still available

long long type

- allows 64 bit integral types
 - signed, unsigned

```
int main()
{
    unsigned long long x = 1;

    for(int i = 0; i < 64; i++)
    {
        cout << i << ":" << x << endl;
        x = x * 2;
    }
}
```

Alternate Function Syntax

- Return types are specified before function definition
 - can cause problems because of single pass compiler

not allowed

```
template< typename L, typename R>  
Ret add(const L &lhs, const R &rhs)  
{  
    return lhs + rhs;  
} //Ret must be the type of lhs+rhs
```

- Alternate syntax now available:

```
template< typename L, typename R>  
auto add(const L &lhs, const R &rhs) -> decltype(lhs+rhs)  
{  
    return lhs + rhs;  
}
```

Controlling Default Methods ...

- **C++ defines default class methods**

- destructor
- default constructor
- copy constructor
- copy assignment operator =

- **... and global operators**

- sequence operator ,
- address-of operator &
- indirection operator *
- member access operator ->
- member indirection operator ->*
- free-store allocation operator new
- free-store deallocation operator delete

**You now have
control over the
generation of these
methods**

... Controlling Default Methods

- For example: How do create a class that doesn't allow copying and assignment?

old way

```
class X {  
public:  
    X() {} // removed by default  
    X(int) { ... }  
private:  
    X(const X&);  
    X& operator=(const X&);  
};
```

new way

```
class X {  
public:  
    X() = default;  
    X(int) { ... }  
    X(const X&) = delete;  
    X& operator=(const X&) = delete;  
};
```

Variadic Template Args ...

- **Boost Function and Tuple classes**
 - really require variadic template parameters
 - work around involves several hacks including
 - repeating code for 1, 2, 3, 4, 5 ... args
 - using macros
- **Variadic template arguments ...**
 - are now part of C++

... Variadic Template Args

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

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

```
int main()
{
    cout << sum(1) << endl;
    cout << sum(1, 2) << endl;
    cout << sum(1, 2, 3) << endl;
    cout << sum(1, 2, 3, 4) << endl;
    cout << sum(1, 2, 3, 4, 5) << endl;

    cout << sum(1.1) << endl;
    cout << sum(1.1, 2.2) << endl;
    cout << sum(1.1, 2.2, 3.3) << endl;
}
```