An Introduction to C++

Part 5

Operators
Function objects
More about templates

Review: what are classes?

- Classes are collections of
  - functions
  - data
  - types
- representing one concept

- These "members" can be split into
  - `public`, accessible interface to the outside
    - should not be modified later!
  - `private`, hidden representation of the concept
    - can be changed without breaking any program using the class
  - this is called "data hiding"
- Objects of this type can be modified only by these member functions -> easier debugging

Special members

- Constructors
  - initialize an object
  - same name as class
- Destructors
  - do any necessary cleanup work when object is destroyed
  - have the class name prefixed by ~
- **Conversion of object A to B**
  - two options:
  - constructor of B taking A as argument
  - conversion operator to type B in A:
    - operator B();
- **Operators**
- Default versions exist for some of these

Operators as functions

- Most operators can be redefined for new classes
- Same as functions, with function name:

```
operator symbol(...)
```

- Example:

```
Matrix A,B,C;
C=A+B;
```

- is converted to
  - either `C.operator=( A.operator+(B));`
  - or `C.operator= (operator+(A,B));`

Assignment operators

- The assignment operators `=`, `+=`, `-=`, `*=`, `/=`, `^=`, `&=`, `|=`, `%=`
  - can only be implemented as member functions
  - should always return a const reference to allow expressions like

```
a=b=c;
a=b+=c;
```

- Example:

```
class Point {
    double x_,y_;
  public:
    const Point& operator+=(const Point& rhs) {
      x_ += rhs.x_;
      y_ += rhs.y_;
      return *this;
    }
};
```

Symmetric operators

- Symmetric operators, e.g. `+`, `-`, … are best implemented as free functions
- Either the simple-minded way

```
Point operator+(const Point& x, const Point& y) {
  Point result(x.x() + y.x(), x.y() + y.y());
  return result;
}
```

- Or the elegant way

```
Point operator+(Point x, const Point& y) {
  return x+= y;
}
```

Extending classes with operators

- Extensions to existing classes can only be implemented as free functions
- Example: extending the iostream library
  - ```
    std::ostream& operator <<(std::ostream& out,
                              const Point& p) {
      out << "( " << p.x() << ", " << p.y() << " )";
      return out;
    }
    ```
- We can now print a `Point`:
  - ```
    Point p;
    std::cout << "The point is " << p << std::endl;
    ```

More comments about operators

- `A a; ++a;` uses
  `const A& A::operator++();`
  or `const A& operator++(A&);`
- `A a; a++;` uses
  `A A::operator++(int);`
  or `A operator++(A&,int);`
  The additional int argument is just to distinguish the two
- `A b; b=a;` uses the assignment
  `const A& A::operator=(const A&);`
- `A b=a;` and `A b(a);` both use the copy constructor
  `A::A(const A&);`

Conversion operators

- conversion of `A -> B` as in:
  - `A a; B b=B(a);`
- can be implemented in two ways
  - constructor `B::B(const A&);`
  - conversion operator `A::operator B();`
- Automatic conversions:
  - `char -> int`
  - `unsigned -> signed`
  - `short -> int -> long`
  - `float -> double -> long double`
  - Integer -> floating point
  - as in: `double x=4;`

Array subscript operator: `operator[]`

- In an array or vector class we want to be use the array subscript syntax:
  - ```
    Array a;
    for (int i=0;i<a.size();++i)
      std::cout << a[i] << std::endl;
    ```
- We need to implement both const and non-const `operator[]`:
  - ```
    class Array {
      public:
        …
        double operator[](unsigned int i) const
        { return p[i];}
        double& operator[](unsigned int i)
        { return p[i];}
      private:
        double* p;
    };
    ```

Pointer operators: `operator*` and `operator->`

- We will get to know classes acting like pointers
  - Iterators
  - Smart pointers (e.g. reference counted or checked pointers)
- In such classes we want to use the pointer syntax
  - `*p`
  - `p->f()`
- We need to implement const and non-const versions of these operators:
  - ```
    class P {
      …
      double* operator->() { return p_;}
      const double* operator->() const { return p_;}
      double& operator*() { return *p_;}
      const double& operator*() const { return *p_;}
    private:
      double* p_;
    };
    ```

The function call operator: `operator()`

- We sometimes want to use an object like a function, e..g
  - ```
    Potential V;
    double distance;
    std::cout << V(distance);
    ```
- This works only if Potential is a function pointer, or if we define the function call operator:
  - ```
    class Potential {
      double operator()(double d) { return 1./d;}
      …
    };
    ```
- Don't get confused by the two pairs of ()()
  - The first is the name of the operator
  - The second is the argument list
    ```

References as return types

- **Warning!** What is wrong?

```
typedef Array<int> IA;
IA& operator+(const IA& x, const IA& y) {
  IA result=x;
  result+=y;
  return result;
}

IA a,b,c;
c=a+b;
```

- Problem: we return reference to temporary object!
  - Very dangerous, will in most cases crash the program
- Correct version copies the result

```
IA operator+(const IA& x, const IA& y) {
  IA result=x;
  x+=y;
  return result;
}
```

Template specialization

- Consider our `Array<T>`
  - An array of size n takes `n*sizeof(T)` bytes
- Consider `Array<bool>`
  - An array of size n takes `n*sizeof(T) = n` bytes
  - An optimized implementation just needs one bit!
  - `Array<bool>(n)` would need only n/8 bytes
- How can we define an optimized version `for bool`?
- Solution: template specialization

```
template <class T>
class Array {
   // generic implementation
   …
   };
```

```
template <>
class Array<bool> {
   //optimized version for bool
   …
   };
```

Traits types

- We want to allow the addition of two arrays:
  - ```
    template <class T>
    Array<T> operator+ (const Array<T>&, const Array<T>&)
    ```
- How do we add two different arrays?
  Array<int> + Array<double> makes sense!
  - ```
    template <class T, class U>
    Array<?> operator +(const Array<T>&, const Array<U>&)
    ```
- What is the result type?
  - We want to calculate with types!
- The solution is a technique called traits. Used quite often
  - numeric_limits traits class for numeric data types
  - can also be used here:
  - ```
    template <class T, class U>
    Array< typename sum_type<T,U>::type >
    operator +(const Array<T>&, const Array<U>&)
    ```

Traits types (continued)

- We want to use traits like
  - ```
    template <class T, class U>
    Array< typename sum_type<T,U>::type >
    operator +(const Array<T>&, const Array<U>&)
    ```
  - The typename keyword is needed with template dependent types
- Definition of `number_traits`:
  - empty template type to trigger error messages if used
    - `template< class T, class U > class sum_type {};`
  - Partially specialized valid templates:
    - `template <class T> struct sum_type<T,T> {typedef T type;};`
  - Fully specialized valid templates:
    - `template <> struct sum_type<double,float> {typedef double type;};`
    - `template <> struct sum_type<float,double> {typedef double type;};`
    - `template <> struct sum_type<float,int> {typedef float type;};`
    - `template <> struct sum_type<int,float> {typedef float type;};`

typename

- The keyword typename is needed here so that C++ knows the member is a type and not a variable or function.

```
template <class T, class U>
Array< typename sum_type<T,U>::type >
operator +(const Array<T>&, const Array<U>&)
```

- This is required to parse the program code correctly – it would not be able to check the syntax otherwise

Old style traits

- In C++98 traits were big "blobs":

```
template<>
struct numeric_limits<int> {
  static const bool is_specialized = true;
  static const bool is_integer = true;
  static const bool is_signed = true;
 ……
};
```

- Later it was realized that this was ugly:
  - A traits class is a "meta function", a function operating on types
  - A blob like numeric limits takes one argument, and returns many different values
  - This is not the usual design for functions!

New style traits

- Since C++03 all new traits are single-valued functions
  - Types are returned as the `type` member:

```
template<class T>
struct sum_type { typedef T type; };

template<>
struct sum_type<int> { typedef double type; };
```

  - Constant values are returned as the `value` member:

```
template<class T>
struct is_integral { static const bool value=false; };

template<>
struct is_integral<int> { static const bool value=true; };
```

Another application of traits

- Imagine an average() function:

```
template <class T>
T average(const Array<T>& v) {
  T sum;
  for (int n=0;n<v.size();++n)
    sum += v[n];
  return sum/v.size();
}
```

- Has problems with Array<int>, as the average is in general a floating point number:
  - v = (1,4,3)
  - Average would be int(8/3)=2
- Solution: traits

- The better version:

```
template <class T>
typename average_type<T>::type
average(const Array<T>& v) {
 typename average_type<T>::type sum;
  for (int n=0;n<v.size();++n)
    sum += v[n];
  return sum/v.size();
}
// the general traits type:
template <class T>
struct average_type {
  typedef T type;
};
// the special cases:
template<>
struct average_type <int> {
  typedef double type;};
…
// repeat for all integer types
```

An automatic solution for all integral types

```
template <class T> struct average_type {
  typedef typename
            helper1<T, std::numeric_limits<T>::is_specialized>::type type;
};
// the first helper:
template<class T, bool F>
struct helper1 { typedef T type };

//  the first helper if numeric_limits is defined: a partial specialization
template<class T>
struct helper1<T,true> {
  typedef typename
            helper2<T, std::numeric_limits<T>::is_integer>::type type;
};
// the second helper:
template<class T, bool F>
struct helper2 { typedef T type };

//  the second helper if the type is integer: a partial specialization
template<class T>
struct helper2<T,true> { typedef double type;}
```

Procedural programming

- ```
  double integrate( double (*f) (double)),
                    double a, double b, unsigned int N)
  {
    double result=0;
    double x=a;
    double dx=(b-a)/N;
    for (unsigned int i=0; i<N; ++i, x+=dx)
      result +=f(x);
    return result*dx;
  }
  ```
- ```
  double func(double x) {return x*sin(x);}
  cout << integrate(func,0,1,100);
  ```
- same as in C, Fortran, etc.

Generic programming

- ```
  template <class T, class F>
  T integrate(F f, T a, T b, unsigned int N)
  {
    T result=T(0);
    T x=a;
    T dx=(b-a)/N;
    for (unsigned int i=0; i<N; ++i, x+=dx)
      result +=f(x);
    return result*dx;
  }
  ```
- ```
  inline double func(double x) {return x*sin(x);}
  std::cout << integrate(func,0.,1.,100);
  ```
- allows inlining!
- works for any type T and F!

Function objects

- Assume a function with parameters: $f(x;\lambda) = \exp(-\lambda x)$
  - ```
    double func(double x, double lambda) {
      return exp(-lambda*x);
    }
    ```
  - cannot be used with integrate template!
- Solution: use a function object
  - ```
    class MyFunc {
      const double lambda;
    public:
      MyFunc(double l) : lambda(l) {}
      double operator() (double x) {return exp(-lambda*x);}
    };
    ```
  - ```
    MyFunc f(3.5)
    integrate(f,0.,1.,1000);
    ```
  - uses object of type MyFunc like a function!
- **Very useful and widely used technique**
    ```