

An Introduction to C++

Part 3

More templates
Function objects
More C++ features

Review of inheritance

◆ Abstract base class

- ◆ represents common interface and functionality

```
◆ class Shape {  
  public:  
    Shape() {};  
    virtual void draw()=0;  
};
```

◆ Concrete derived class

- ◆ implements functionality for given instance

```
◆ class Triangle : public Shape {  
  public:  
    void draw(); // draws triangle  
};
```

◆ Usage in function

```
◆ print_figure(Shape& s);
```

Relationship to templates

- ◆ Object Oriented Programming:

- ◆ `void show(Shape& s) {`
`s.draw();`
`}`

- ◆ Object needs to be derived from Shape

- ◆ Concrete type decided at runtime

- ◆ Generic programming:

- ◆ `template <class T> void show(T& s) {`
`s.draw();`
`}`

- ◆ Object needs to have a draw function

- ◆ Concrete type decided at compile time

When to use which?

- ◆ Generic programming allows inlining

- ◆ faster code

- ◆ Object oriented programming more flexible

- ◆ how to draw an Array of shapes?

```
void show(Array<Shape*> a) {
    for (int i=0; i<a.size(); ++i)
        a[i]->draw();
}
```

- ◆ This works for array of mixed shapes, e.g. squares, triangles, ...

Templates vs. inheritance

- | | |
|--|--|
| <ul style="list-style-type: none"> ◆ Object oriented programming ◆ uses inheritance ◆ decision at run-time ◆ works for objects derived from the common base ◆ one function created for the base class -> saves space ◆ virtual function call needs lookup in type table -> slow ◆ extension possible using only definition of base class
 ◆ useful for application frameworks, user interfaces, high level structures | <ul style="list-style-type: none"> ◆ Generic programming ◆ uses templates ◆ decision at compile-time ◆ works for objects having the right members ◆ a new function created for each class used -> wastes space ◆ no virtual function call, can be inlined -> fast ◆ extension needs definitions and implementations of all functions
 ◆ useful for small, low level constructs and generic algorithms |
|--|--|

Calling C functions

- ◆ C functions are identified by **name**
- ◆ C++ functions by **name & parameter list**
 - ◆ needed because of function overloading

- ◆ Consequence
 - ◆ A C function


```
float square(float);
```
 - ◆ is different from the same C++ function!

- ◆ How to declare a C function: extern "C"
 - ◆

```
extern "C" float square(float);
```
 - ◆

```
extern "C" {
    void f1();
    void f2();
}
```

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
 ...
};

```

## Integer template parameters

---

- ◆ Want to calculate  $x^n$
- ◆ recursion function:
 

```

template <class T>
T pow(T x, unsigned int n) {
 return (n ?
 x*pow(x,n-1) : T(1));
}

```

  - ◆ slow because of recursion
- ◆ normal function:
 

```

template <class T>
T pow(T x, unsigned int n) {
 T r=T(1);
 while(n--) r*=x;
 return r;
}

```

  - ◆ slow because of loop

## Integer template parameters

---

- ◆ Template function:

- ◆ `template <class T, int n>`  
`T pow(T x) {`  
`return x*pow<T,n-1>(x);}`
- ◆ `template <class T>`  
`T pow<T,1>(T x) {return x;}`
- ◆ `template <class T>`  
`T pow<T,0>(T x) {return T(1);}`

- ◆ is fast, as recursion evaluated at compile time.

- ◆ The code

- ◆ `pow<double,3>(x) ->`
- ◆ `x*pow<double,2>(x) ->`
- ◆ `x*x*pow<double,1>(x) ->`
- ◆ `x*x*x`

- ◆ no loop! no function call!

- ◆ More optimization: `pow<T,4>(x)` same as `pow<T,2>(pow<T,2>(x))`

## Member templates

---

- ◆ Consider the complex class

- ◆ We want to allow to add `complex<float>` to `complex<double>`:

- ◆ `complex<double> x(1.,0.);`  
`complex<float> y(0.,1.);`  
`x+=y;`
- ◆ `template <class T> class complex {`  
`...`  
`template <class X> complex<T>& operator +=(complex<X>);`  
`};`
- ◆ `template<class T> template <class X>`  
`complex<T>& complex<T>::operator+=(complex<X>) {`  
`...`  
`};`

- ◆ This feature is called “**member templates**”

- ◆ supported by only a few modern compilers

## 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?

- ◆ The solution is a technique called traits. Used quite often

- ◆ character traits for narrow and wide strings, streams, ...
- ◆ numeric\_limits traits class for numeric data types
- ◆ can also be used here:

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

## Traits types (continued)

---

- ◆ We want to use traits like

```
◆ template <class T, class U>
 Array< typename number_traits<T,U>::sumtype >
 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 number_traits {}
```

- ◆ specialized valid templates, containing sum\_type:

```
◆ template <class T> class number_traits<T,T>
 {typedef T sumtype;};
◆ template <> class number_traits<double,float>
 {typedef double sumtype;};
◆ template <> class number_traits<float,double>
 {typedef double sumtype;};
◆ template <> class number_traits<float,int>
 {typedef float sumtype;};
◆ template <> class number_traits<int,float>
 {typedef float sumtype;};
```

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

## The numerical integration exercise

- ◆ The numerical integration exercise demonstrates all four programming styles:
  - ◆ 1st part: **procedural programming**
  - ◆ 2nd part: **modular programming**
    - ◆ We built a library
  - ◆ 3rd part (this week): **object oriented programming**
    - ◆ We made an object
  - ◆ 4th part (next week): **generic programming**
    - ◆ We will use templates
- ◆ After you have coded all four versions, perform benchmarks
  - ◆ Which version is fastest?
  - ◆ Which version is the most flexible?

## Integrating a function

---

- ◆ is done by replacing the integral by a finite sum

- ◆ rectangles

$$\int_a^b f(x) dx = \frac{b-a}{N} \sum_{i=1}^N f\left(a + i \frac{b-a}{N}\right) + O(1/N)$$

- ◆ trapezes

$$\int_a^b f(x) dx = \frac{b-a}{N} \left( \frac{1}{2} f(a) + \sum_{i=1}^{N-1} f\left(a + i \frac{b-a}{N}\right) + \frac{1}{2} f(b) \right) + O(1/N^2)$$

- ◆ parabola (Simpson rule)

$$\int_a^b f(x) dx = \frac{b-a}{3N} \left( f(a) + \sum_{i=1}^{N-1} (3 - (-1)^i) f\left(a + i \frac{b-a}{N}\right) + f(b) \right) + O(1/N^4)$$

- ◆ for higher order schemes and adaptive integrators review your numerics course or look into text books

## Functional 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.

## Object oriented programming

---

```

◆ Class Integrator { // base class implements integration
 public:
 Integrator() {}
 double integrate(double a, double b, unsigned int n);
 virtual double f(double)=0;
};

◆ class MyFunc : public Integrator { // derived class
 public:
 MyFunc() {}
 double f(double x) {return x*sin(x);} //implements function
};

◆ MyFunc f;
 f.integrate(0,1,1000);

```

## 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);}
 cout << integrate(func,0.,1.,100);

```

- ◆ allows inlining!
- ◆ works for any type T!
- ◆ For more tricks see Todd Veldhuizen's notes

## Function objects

- ◆ Assume a function with parameters:  $f(x, I) = \exp(-Ix)$

- ◆ `double func(double x, double lambda) {  
    return exp(-lambda*x);  
}`

- ◆ cannot be used with integrate template!

- ◆ Solution: use a **function object**

- ◆ `class MyFunc {  
    double lambda;  
public:  
    MyFunc(double l) : lambda(l) {}  
    double operator() (double x) {return exp(-lambda*x);}  
};`

- ◆ `integrate(MyFunc(lam), 0., 1., 1000);`

- ◆ uses object of type MyFunc like a function!

- ◆ **Very useful and widely used technique**

## OOP/Functional vs. Generic

- |                                                               |                                                                      |
|---------------------------------------------------------------|----------------------------------------------------------------------|
| ◆ Object Oriented Programming or Functional Programming       | ◆ Generic Programming (using templates)                              |
| ◆ + only one integration function generated for all functions | ◆ - separate integration function for each function to be integrated |
| ◆ + function need <i>not</i> be known at compile time         | ◆ - function needs to be known at compile time                       |
| ◆ - Virtual function call has extra overhead                  | ◆ + allows inlining                                                  |
| ◆ - function objects need to be derived from base class.      | ◆ + allows use of function objects                                   |
| ◆ C, Fortran allow only this way                              | ◆ C++ allows both, more flexibility                                  |