UNIVERSITY OF MANITOBA

Efficient Techniques for Numeric Integration

Lecture 1

ECE7650-T13: C++ and Object-Oriented Numeric Computing for Engineers

Date: Marth 11th, 2016

Electrical and Computer Engineering

UNIVERSITY OF MANITOBA

Outline of Lecture 1

- Recall the Integrations
- Function Object Approach
- Function Pointer as Template Parameter
- Dot Product and Expression Templates
- Dot Product and Template Metaprograms

UNIVERSITY OF MANITOBA

Outline

- **Recall the Integrations**
- Function Object Approach
- Function Pointer as Template Parameter
- Dot Product and Expression Templates
- Dot Product and Template Metaprograms

UNIVERSITY OF MANITOBA

Recall the Integrations

```
typedef double (*pfn)(double);  // define pfn for integrand
double trapezoidal(double a, double b, pfn f, int n);
double simpson(double a, double b, pfn f, int n);

double square(double d) { return d*d; }
double trapezoidal(double a, double b, pfn f, int n) {
  double h = (b - a)/n;        // size of each subinterval
  double sum = f(a)*0.5;
  for (int i = 1; i < n; i++) sum += f(a + i*h);
  sum += f(b)*0.5;
  return sum*h;
}
```

In chapter 3, the Trapezoidal Rules and Simpson's Rule, are coded using traditional procedural programming style, while in chapter 5, a class is used for encapsulation. These programs work fine except for run-time efficiency, when they are called frequently.

UNIVERSITY OF MANITOBA

Recall the Integrations

? How can we improve it?

Templates are compile time feature. It will reduce the runtime.

This section presents several efficient techniques for numeric integration. These techniques include using function objects instead of passing function pointers as arguments to function calls, using function pointers as template parameters, expression templates, and template metaprograms. These techniques are very general and also apply to many other situations.

Note: These advanced techs (by using templates) are used to improve for run-time performance and generality.

Outline

- Recall the Integrations
- **Function Object Approach**
- Function Pointer as Template Parameter
- Dot Product and Expression Templates
- Dot Product and Template Metaprograms

UNIVERSITY OF MANITOBA

Function Object Approach

As pointed out in §7.2 and §7.6, using templates and function objects can improve run-time efficiency for programs that traditionally require passing pointer-to-functions as arguments. Using a template and a function object, the function *trapezoidal()*, for numerically evaluating $\int_a^b f(x)dx$, can be written and used as

```
template<class Fo>
double trapezoidal(double a, double b, Fo f, int n) {
  double h = (b - a)/n;          // size of each subinterval
  double sum = f(a)*0.5;
  for (int i = 1; i < n; i++) sum += f(a + i*h);
  sum += f(b)*0.5;
  return sum*h;
}
class integrand {       // define a class for my integrand
public:
  double operator()(double x) { return exp(-x*x); }
};
int main() {
  cout << trapezoidal(0, 1, integrand(), 100) << '\n';
}
```

UNIVERSITY OF MANITOBA

Function Object Approach

Recall that an object of a class with the function call operator () overloaded is called a *function object*. Thus the object *integrand*(), instead of the class name *integrand*, is passed to the function call *trapezoidal*(). The compiler will then instantiate *trapezoidal*(), substitute *integrand* for the template parameter *Fo*, which results in *integrand::operator*() being inlined into the definition of *trapezoidal*(). Note that *trapezoidal*() also accepts a pointer-to-function as its third argument, but this would be hard to be inlined and may not improve the run-time.

UNIVERSITY OF MANITOBA

Function Object Approach

Put it in a more general form :

```
template<class T, class Fo>
T trapezoidal(T a, T b, Fo f, int n) {
  T h = (b - a)/n;          // size of each subinterval
  T sum = f(a)*0.5;
  for (int i = 1; i < n; i++) sum += f(a + i*h);
  sum += f(b)*0.5;
  return sum*h;
}

template<class T> class integrand2 {
public:                     // define a class for my integrand
  T operator()(T x) { return exp(-x*x); }
};

int main() {
  cout << trapezoidal(0.0, 1.0, integrand2<double>(), 100);
}
```

With this form, integrals such as $\int_0^1 e^{-x^2} dx$ can be integrated by the Trapezoidal Rule in double, float, long double, int or other precisions.

Outline

- Recall the Integrations
- Function Object Approach
- **Function Pointer as Template Parameter**
- Dot Product and Expression Templates
- Dot Product and Template Metaprograms

Function Pointer as Template Parameter

The technique here to improve run-time efficiency still uses a function pointer, but in a very different way, in which the function pointer is used as a template parameter, Thus, it canbe inlined to avoid function call overheads. It can be illustrated by implementing the Trapezoidal Rule:

```
template<double F(double)>    // F is a template parameter
double trapezoidal(double a, double b, int n) {
  double h = (b - a)/n;       // size of each subinterval
  double sum = F(a)*0.5;
  for (int i = 1; i < n; i++) sum += F(a + i*h);
  sum += F(b)*0.5;
  return sum*h;
}

double myintegrand(double x) {
  return exp(-x*x);
}

int main() {
  cout << trapezoidal<myintegrand>(0, 1, 100) << '\n';
}
```

Function Pointer as Template Parameter

With this approach, the integrand is still implemented as a function. But it is not passed as an argument to *trapezoidal().* Instead, it is taken as a template parameter, which will be inlined. It can be generalized into:

```
template<class T, T F(T)>
T trapezoidal(T a, T b, int n) {
  T h = (b - a)/n;        // size of each subinterval
  T sum = F(a)*0.5;
  for (int i = 1; i < n; i++) sum += F(a + i*h);
  sum += F(b)*0.5;
  return sum*h;
}

template<class T> T myintegrand2(T x) {
  return exp(-x*x);
}

int main() {
  cout << trapezoidal<double, myintegrand2>(0,1,100) << '\n';
}
```

Outline

- Recall the Integrations
- Function Object Approach
- Function Pointer as Template Parameter
- **Dot Product and Expression Templates**
- Dot Product and Template Metaprograms

Using Dot Product and Expression Templates

New Problem:

For integrals such as $\int_a^b [f(x) + g(x)]dx$ and $\int_a^b f(x)g(x)h(x)dx$, the integrand is a product (or summation) of different functions such as f, g, and h, and the C++ programs presented in previous sections can not be directly applied, since code such as

```
double result0 = trapezoidal(0, 1, f + g, 100);
double result2 = trapezoidal(0, 1, f*g*h, 100);
```

is illegal.

Using Dot Product and Expression Templates

One Solution:

One way is to write another function that returns the product or sum of some functions.

```
double F(double x, double f(double), double g(double)) {
  return f(x) + g(x);
}

double G(double x, double f(double), double g(double),
         double h(double)) {
  return f(x)*g(x)*h(x);
}
```

Using Dot Product and Expression Templates

Not Good:

But the original integration function

```
template<class Fo>
double trapezoidal(double a, double b, Fo f, int n);
```

can no longer be applied to F or G (due to the complicated prototypes of F and G), and thus several versions of it are needed. For a typical finite element analysis code, the integration of the product (or sum) of many functions (such as basis functions and their derivatives, coefficient functions and their derivatives) need be evaluated, and this approach would lead to many overloaded integration functions that are hard to manage and maintain.

Using Dot Product and Expression Templates

How to solve it?

1) Vector and operator overload

In this subsection, dot products are used to evaluate numeric integration. As an example, consider a typical Gauss quadrature:

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} w_i f(x_i), \qquad (7.1)$$

2) Issue "Vs = V1 + V2 + V3;" -→ expression templates

The idea is to defer the evaluations of intermediate sub-operations in a composite operation until the end of the operation. (As we did in Chapter 6.5)

UNIVERSITY OF MANITOBA

Using Dot Product and Expression Templates

1) Vector and operator overload

In this subsection, dot products are used to evaluate numeric integration. As an example, consider a typical Gauss quadrature:

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} w_i f(x_i), \tag{7.1}$$

where $[x_0, x_1, \ldots, x_{n-1}]$ are given Gauss points and $[w_0, w_1, \ldots, w_{n-1}]$ are the corresponding weights; see Exercise 5.10.10. If vector *wgts* contains the weights w_i and vector *fxi* contains the function values at Gauss points, then the dot product function can be called to give the approximate integral:

```
double dot(const Vtr& u, const Vtr& v) {
  double dotprod = 0;
  int n = u.size();
  for (int i = 0; i < n; i++) dotprod += u[i]*v[i];
  return dotprod;
}

Vtr fxi(n);
for (int i = 0; i < n; i++) fxi[i] = f(x[i]);
double d = dot(fxi, wgts);  // dot() evaluates integrals
```

UNIVERSITY OF MANITOBA

Using Dot Product and Expression Templates

1) Vector and operator overload

Here *Vtr* is a vector class (e.g., the standard library *vector<double>* or the class *Vcr* defined in Chapter 6). When an integrand is the product of many functions such as $f * g * h$, a vector-multiply function can be defined:

```
Vtr operator*(const Vtr& u, const Vtr& v) {
  int n = u.size();
  Vtr w(n);
  for (int i = 0; i < n; i++) w[i] = u[i]*v[i];
  return w;
}

Vtr fghxi = fxi*gxi*hxi;
double d0 = dot(fghxi, wgts);  // evaluate integral of fgh
double d2 = dot(fxi, wgts);    // evaluate integral of f
```

Using Dot Product and Expression Templates

1) Vector and operator overload

where fxi, gxi, and hxi are the vectors consisting of the function values of f, g, and h, respectively, evaluated at the Gauss points $[x_0, x_1, \ldots, x_{n-1}]$ of a given Gauss quadrature. This is a traditional way to evaluate integrals with integrands as products (or sums or a combination of them) of many different functions. It works fine and is very convenient to use, but performs very poorly due to the extra work in computing the loop for $fxi*gxi$, storing the result into a temporary vector tmp, computing another loop to obtain $tmp * hxi$, and storing it in another temporary vector; see §6.5 for similar situations.

2) Issue "Vs = V1 + V2 + V3*2;" -→ expression templates ates

Expression templates can be applied to improve run-time efficiency of such vector multiplications by deferring intermediate operations and avoiding temporary objects. It is similar to the technique discussed in §6.5, but is more general. First define a class to hold left operand, operator, and right operand in an operation such as $X * Y$:

```
template<class LeftOpd, class Op, class RightOpd>
struct LOR {
  LeftOpd fod;      // store left operand
  RightOpd rod;     // store right operand

  LOR(LeftOpd p, RightOpd r): fod(p), rod(r) { }

  double operator[](int i) {
    return Op::apply(fod[i], rod[i]);
  }
};
```

Here the template parameter Op refers to a class whose member $apply()$ defines a specific operation on elements of the left and right operands. It defers intermediate evaluations and thus avoids temporary objects and loops. Evaluation of the operation such as $X * Y$ takes place only when the operator [ ] is called on an object of type LOR.

2) Issue "Vs = V1 + V2 + V3*2;" -→ expression templates lates

Then define a vector class, in which the operator = is overloaded that forces the evaluation of an expression:

```
class Vtr {
  int lenth;
  double* vr;
public:
  Vtr(int n, double* d) { lenth = n; vr = d; }

  // assign an expression to a vector and
  // do the evaluation
  template<class LeftOpd, class Op, class RightOpd>
  void operator=(LOR<LeftOpd, Op, RightOpd> exprn) {
    for (int i = 0; i < lenth; i++) vr[i] = exprn[i];
  }

  double operator[](int i) const { return vr[i]; }
};
```

Notice the function call *exprn*[ ] forces the operation *Op*::*apply*() to evaluate elements of the left and right operands.

2) Issue "Vs = V1 + V2 + V3*2;" -→ expression templates lates

Next define a class on how left and right operands are going to be evaluated:

```
// define multiply operation on elements of vectors
struct Multiply {
  static double apply(double a, double b) {
    return a*b;
  }
};
```

Finally overload the operator $*$ for vector multiplication:

```
template<class LeftOpd>
LOR<LeftOpd, Multiply, Vtr> operator*(LeftOpd a, Vtr b) {
  return LOR<LeftOpd, Multiply, Vtr>(a,b);
}
```

The multiplication is actually deferred. Instead a small object of type LOR is returned. For example, $X * Y$ gives an object of LOR (without actually multiplying X and Y), where Y is of type Vtr, but X can be of a more general type such as Vtr and LOR.

Using Dot Product and Expression Templates

3) An example

```
int main() {
  double a[] = { 1, 2, 3, 4, 5};
  double b[] = { 6, 7, 8, 9, 10};
  double c[] = { 11, 12, 13, 14, 15};
  double d[5];

  Vtr X(5, a), Y(5, b), Z(5, c), W(5, d);
  W = X * Y * Z;
}
```

UNIVERSITY OF MANITOBA

Using Dot Product and Expression Templates

3) An example

When the compiler sees the statement $W = X * Y * Z$; it constructs the object $LOR<X, Multiply, Y>$ from $X * Y$ and constructs the object

$$LOR< LOR<X, Multiply, Y>, Multiply, Z>$$

from $LOR<X, Multiply, Y> * Z$ for the expression $X * Y * Z$. When it is assigned to object W, the operation $Vtr::operator=$ is matched as

```
W.operator=(LOR< LOR<X, Multiply, Y>, Multiply, Z> exprn) {
  for (int i = 0; i < lenth; i++) vr[i] = exprn[i];
}
```

where $exprn[i]$ is then expanded by inlining the $LOR::operator[\,]$:

```
Multiply::apply(LOR<Vtr,Multiply,Vtr>(X,Y)[i], Z[i]);
```

which is further expanded into

```
Multiply::apply(X[i], Y[i]) * Z[i]
=  X[i] * Y[i] * Z[i].
```

The final result of $W = X * Y * Z$ is:

```
for (int i = 0; i < lenth; i++) W[i] = X[i] * Y[i] * Z[i];
```

Using Dot Product and Expression Templates

4) Conclusion

Thus there are no temporary vector objects or extra loops.

Similarly, vector multiplication $V = W * X * Y * Z$ will be finally expanded by the compiler into

```
for (int i = 0; i < lenth; i++)
  V[i] = W[i] * X[i] * Y[i] * Z[i];
```

That is, the expression template technique efficiently computes the product of an arbitrary number of vectors with a single loop and without temporary vector objects.

Outline

- Recall the Integrations
- Function Object Approach
- Function Pointer as Template Parameter
- Dot Product and Expression Templates
- **Dot Product and Template Metaprograms**

UNIVERSITY OF MANITOBA

Dot Product and Template Metaprograms

In §7.7.3, numeric integration is evaluated through dot products, where vector multiplication can be efficiently computed by using expression templates. A typical numeric integration formula (see Exercise 5.10.11 and §7.7.3) has very few terms in the summation, which implies that dot products of small vectors are usually evaluated. A dot product function is normally written using a loop:

```
double dot(const Vtr& u, const Vtr& v) {
  double dotprod = 0;
  int n = u.size();
  for (int i = 0; i < n; i++) dotprod += u[i]*v[i];
  return dotprod;
}
```

where the size of the vectors may not have to be known at compile-time.

Dot Product and Template Metaprograms

Example 1

For vectors of a small size (say 4), an alternative definition:

```
inline double dot4(const Vtr& u, const Vtr& v) {
  return u[0]*v[0] + u[1]*v[0] + u[2]*v[2] + u[3]*v[3];
}
```

can boost performance, since it removes loop overhead and low-level parallelism can be easily done on the summation operation. Besides, it is easier to make this small function to be inlined (reducing function-call overhead) and its data to be registerized. All these factors can make *dot*4() much more efficient than the more general *dot*() for vectors of size 4.

To improve performance, template metaprograms can be used so that the dot products of small vectors are expanded by the compiler in the form of *dot*4().

UNIVERSITY OF MANITOBA

Dot Product and Template Metaprograms

Example 2

To illustrate the idea of template metaprograms, consider the Fibonacci numbers defined by a recursive relation: $f_0 = 0, f_1 = 1$, $f_n = f_{n-1} + f_{n-2}$, for $n = 2, 3, \ldots$. They can be recursively computed by the compiler:

```
template<int N> struct fib{
  enum { value = fib<N-1>::value + fib<N-2>::value};
};

template<> struct fib<1>{
  enum { value = 1 };
};

template<> struct fib<0>{
  enum { value = 0 };
};

const int f3 = fib<3>::value;
const int f9 = fib<9>::value;
```

The compiler can generate that f3 = 2 and f9 = 34. The recursive template behaves like a recursive function call (but at compile-time) and the specializations fib<1> and fib<0> stop the recursion.

Dot Product and Template Metaprograms

Template metaprogram

This recursive template technique can be generalized to compute dot products. First define a class for small vectors:

```
template<int N, class T>
class smallVtr {
  T vr[N];        // array of size N and type T
public:
  T& operator[](int i) { return vr[i]; }
};
```

Then define a dot product function that calls a template metaprogram:

```
template<int N, class T>
inline T dot(smallVtr<N,T>& u, smallVtr<N,T>& v) {
  return metaDot<N>::f(u,v);
}
```

Dot Product and Template Metaprograms

Template metaprogram

where the template metaprogram *metaDot* can be defined as

```
template<int M> struct metaDot {
  template<int N, class T>
  static T f(smallVtr<N,T>& u, smallVtr<N,T>& v) {
    return u[M-1]*v[M-1] + metaDot<M - 1>::f(u,v);
  }
};

template<> struct metaDot<1> {
  template<int N, class T>
  static T f(smallVtr<N,T>& u, smallVtr<N,T>& v) {
    return u[0]*v[0];
  }
};
```

The specialization *metaDot*<1> stops the recursion.

Dot Product and Template Metaprograms

Example

```
int main() {
  smallVtr<4, float> u, v;
  for (int i = 0; i < 4; i++) {
    u[i] = i + 1;
    v[i] = (i + 2)/3.0;
  }
  double d = dot(u,v);
  cout << " dot = " << d << '\n';
}
```

It is this compile-time expanded code that gives better performance than a normal function call with a loop inside.

Where the inlined function call dot(u,v) is expanded during compilation as:

```
dot(u,v)
= metaDot<4>::f(u,v)
= u[3]*v[3] + metaDot<3>::f(u,v)
= u[3]*v[3] + u[2]*v[2] + metaDot<2>::f(u,v)
= u[3]*v[3] + u[2]*v[2] + u[1]*v[1] + metaDot<1>::f(u,v)
= u[3]*v[3] + u[2]*v[2] + u[1]*v[1] + u[0]*v[0]
```

UNIVERSITY OF MANITOBA

Dot Product and Template Metaprograms

Conclusion

This template metaprogram technique could result in code that is several times faster than a normal function call when computing dot products of small vectors.

In large-scale applications, it is not enough to compute correctly, but also efficiently and elegantly. The elegance of a program should also include extendibility, maintainability, and readability, not just the look of the program. The code for dot products of small vectors using the template metaprogram technique is also elegant in the sense that it can be easily extended to other situations, for example, dot products of complex vectors, and can be maintained easily. Writing it in FORTRAN or C styles would possibly result in tens of versions (dot products of vectors of sizes 2, 3, 4, 5, and so on, and for real and complex small vectors in different precisions). The Trapezoidal Rule itself is also written in efficient and elegant ways in this section and many other functions such as Simpson's Rule can be as well; see Exercise 7.9.11.

UNIVERSITY OF MANITOBA

Polynomial Interpolation

Lecture 2

ECE7650-T13: C++ and Object-Oriented Numeric Computing for Engineers

Date: March 11th, 2016

Electrical and Computer Engineering

UNIVERSITY OF MANITOBA

Outline of Lecture 2

- Introduction
- Lagrange Form
- Newton Form

UNIVERSITY OF MANITOBA

Outline

- **Introduction**
- Lagrange Form
- Newton Form

Introduction

- A Example
  - Given a set of $n+1$ values $\{y_i\}_{i=0}^{n}$ of a function $f(x)$ at $n+1$ distinct points $\{x_i\}_{i=0}^{n}$, where n is a positive integer, how can one approximate $f(x)$ for $x \neq x_i, i = 0,1,\ldots,n$?
- Such a problem arises in many applications in which the $n+1$ pairs $\{x_i, y_i\}$ may be obtained from direct measurement, experimental observations, or an expensive computation but function values $f(x)$ at other points are needed.

Introduction : continued

- Purposes
  - Some of the values are needed but unknown. (approximate / predict)
  - Approximate functions with simpler ones. (usually polynomials or piecewise polynomials)

- A polynomial is an expression consisting of variables and coefficients, that involves only the operations of addition, subtraction, multiplication, and non-negative integer exponents.

Introduction : continued

- A polynomial in a single indeterminate can be written in the form:

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0,$$

- This section deals with polynomial interpolations, in which a polynomial $p_n(x)$ of degree n is constructed to approximate $f(x)$ that satisfies the conditions $f(x_i) = p_n(x_i), i = 0,1, \ldots, n.$

Introduction : Questions

- How to *represent* the interpolating polynomial $p_n(x)$?
- How to determine the associated *coefficients?*
- After we have obtained the coefficients, how can the interpolation be *evaluated* (at other values of *x*) with efficiency?

UNIVERSITY OF MANITOBA

Outline

- Introduction
- **Lagrange Form**
- Newton Form

Lagrange Form

- The approximating polynomial can be written in the Lagrange Form:

$$p_n(x) = y_0 L_0^{(n)}(x) + y_1 L_1^{(n)}(x) + \cdots + y_n L_n^{(n)}(x) = \sum_{i=0}^{n} y_i L_i^{(n)}(x)$$

- each $L_i^{(n)}(x)$ is a polynomial of degree n and depends only on the interpolation nodes $\{x_i\}_{i=0}^n$, but not on the given function values $\{y_i\}_{i=0}^n$. They are defined as

$$L_i^{(n)}(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}, \qquad i = 0, 1, \ldots, n-1.$$

Lagrange Form : continued

- This is , each $L_i^{(n)}(x)$ is the product of all factors $\frac{(x - x_j)}{(x_i - x_j)}$ for $i \neq j$.
- They have the properties: Satisfy the Condition

$$L_i^{(n)}(x_j) = 0 \qquad \text{for } j \neq i, \qquad \text{and } L_i^{(n)}(x_i) = 1.$$

- Thus $p_n(x)$ agree with $f(x)$ at the given interpolation nodes $\{x_i\}_{i=0}^{n}$. And function values for the else x could be approximated by $p_n(x)$.

Liner interpolation

- When $n = 1$, the interpolation polynomial is a line passing through (x_0, y_0) and (x_1, y_1), and has the form:

$$p_1(x) = L_0^{(1)}(x)y_0 + L_1^{(1)}(x)y_1 = \frac{x - x_1}{x_0 - x_1}y_0 + \frac{x - x_0}{x_1 - x_0}y_1.$$

- This is the so-called linear interpolation, which is used frequently in many scientific works.

Quadratic polynomial

- When $n = 2$, the $p_2(x)$ is a quadratic polynomial passing through (x_0, y_0), (x_1, y_1), and (x_2, y_2), and has the form:

$$p_2(x) = \sum_{i=0}^{2} L_i^{(2)}(x) y_i$$

$$= \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} y_0 + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} y_1 + \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} y_2$$

UNIVERSITY OF MANITOBA

Lagrange Form : continued

- Since some applications require more accuracy than others, a template function can be written so that a user can conveniently choose single, double, long double, or other precisions.

```
template<class T>
T lagrange(const vector<T>& vx, const vector<T>& vy, T x) {
  int n = vx.size() - 1;
  T y = 0;
  for (int i = 0; i <= n; i++) {
    T temp = 1;
    for (int j = 0; j <= n; j++)
      if (j != i) temp *= (x - vx[j])/(vx[i] - vx[j]);
    y += temp*vy[i];
  }
  return y;
}
```

UNIVERSITY OF MANITOBA

Lagrange Form : A Example

- Consider the interpolation problem that gives $x_i = 1 + i/4.0$ and $y_i = e^{x_i}$, for $i = 0,1,2,3$. the function template can then be userd to find an approximation function value at $x = 1.4$.

```
int main() {
  const int n = 4;
  vector<float> px(n);
  vector<float> py(n);
  for (int i = 0; i < n; i++) {
    px[i] = 1 + i/4.0; py[i] = exp( px[i] );
  }
  float x = 1.4;
  float approximation = lagrange(px, py, x);
}
```

Result: Compared to exact function value $e^{1.4}$, the error of this approximation is 0.0003497.

UNIVERSITY OF MANITOBA

Outline

- Introduction
- Lagrange Form
- **Newton Form**

Newton Form

- The Newton form is more efficient and seeks $p_n(x)$ *as:*

$$p_n(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + \cdots + c_n(x - x_0)\cdots(x - x_{n-1})$$

- $c_0, c_1, \ldots, c_n$ are constants to be determined.

- For example, the linear and quadratic interpolation polynomials are

$$p_1(x) = c_0 + c_1(x - x_0),$$
$$p_2(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1)$$

Newton Form : Evaluate

- Firstly, we talk about how Newton's form can be evaluated efficiently, assuming all the coefficients are known.
- Write $p_n(x)$ into a nested multiplication form:

$$p_n(x) = c_0 + d_0(c_1 + d_1(c_2 + \cdots + d_{n-3}(c_{n-2} + d_{n-2}(c_{n-1} + d_{n-1}(c_n))) \cdots))$$

where $d_0 = x - x_0, d_1 = x - x_1, \ldots, d_{n-1} = x - x_{n-1}$

- Introduce the new notation:

$$\begin{aligned} u_n &= c_n; \\ u_{n-1} &= c_{n-1} + d_{n-1}u_n; \\ u_{n-2} &= c_{n-2} + d_{n-2}u_{n-1}; \\ &\vdots \\ u_1 &= c_1 + d_1u_2; \\ u_0 &= c_0 + d_0u_1; \end{aligned}$$

Newton Form : Evaluate

- Then $p_n(x) = u_0$. This procedure can be written into an algorithm:

$u \leftarrow c_n;$
for $(i = n-1, n-2, \ldots, 0)$ {
  $u \leftarrow c_i + d_i * u;$ // that is: $u \leftarrow c_i + (x - x_i) * u$
}
return $u;$

- This is the so-called Horner's algorithm, which is very efficient in evaluating polynomials.

Newton Form : Coefficients

- The requirement $p_n(x_i) = f(x_i)$ leads to

$$
\begin{aligned}
f(x_0) &= p_n(x_0) = c_0, \\
f(x_1) &= p_n(x_1) = c_0 + c_1(x_1 - x_0), \\
f(x_2) &= p_n(x_2) = c_0 + c_1(x_2 - x_0) + c_2(x_2 - x_0)(x_2 - x_1), \\
&\vdots
\end{aligned}
$$

- Thus, $c_0 = f(x_0)$ and $c_1 = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$

Newton Form : Coefficients

- Define the divided differences of order i as:
  - Order 0 : $f[x_i] = f(x_i)$
  - Order 1 : $f[x_i, x_j] = \frac{f[x_j] - f[x_i]}{x_j - x_i}, j \neq i$
  - Order j : $f[x_i, x_{i+1}, \dots, x_{i+j}]$

$$= \frac{f[x_{i+1}, x_{i+2}, \dots, x_{i+j}] - f[x_i, x_{i+1}, \dots, x_{i+j-1}]}{x_{i+j} - x_i}$$

Newton Form : Coefficients

- Then we could find that:
  - Order 0 : $c_0 = f[x_0]$,
  - Order 1 : $c_1 = f[x_0, x_1] = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$
  - Order j : $c_2 = \frac{f(x_2) - f(x_0) - c_1(x_2 - x_0)}{(x_2 - x_0)(x_2 - x_1)}$

$$= \frac{\frac{f(x_2) - f(x_1)}{(x_2 - x_1)} + \frac{f(x_1) - f(x_0) - c_1(x_2 - x_0)}{(x_2 - x_1)}}{(x_2 - x_0)}$$

$$= \frac{f[x_1, x_2] + \frac{c_1(x_1 - x_0) - c_1(x_2 - x_0)}{(x_2 - x_1)}}{(x_2 - x_0)} =$$

$$\frac{f[x_1, x_2] - f[x_1, x_0]}{(x_2 - x_0)} = f[x_0, x_1, x_2]$$

Newton Form : Coefficients

Similarly, it can be shown that $c_i = f[x_0, x_1, \ldots, x_i]$ for $i = 0, 1, \ldots, n$. That is, all the coefficients c_i can be expressed as divided differences and the interpolation polynomial $p_n(x)$ is:

$$p_n(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + \cdots$$
$$+ f[x_0, x_1, \ldots, x_n](x - x_0)(x - x_1) \cdots (x - x_{n-1}).$$

If a table of function values $(x_i, f(x_i))$ is given, then a table of divided differences can be constructed recursively column by column as indicated in Table 7.1 for the case of $n = 3$.

Newton Form : Differences

x_0	$f[x_0]$	$f[x_0, x_1]$	$f[x_0, x_1, x_2]$	$f[x_0, x_1, x_2, x_3]$
x_1	$f[x_1]$	$f[x_1, x_2]$	$f[x_1, x_2, x_3]$	
x_2	$f[x_2]$	$f[x_2, x_3]$		
x_3	$f[x_3]$			

TABLE 7.1. A table of divided differences in the case of $n = 3$.

In particular, the first two columns list the given x and y coordinates of the interpolation points, divided differences of order 1 are computed and put in the third column, divided differences of order 2 are computed and put in the fourth column, and so on. When the table is completed, the divided differences in the top row will be used to compute $p_n(x)$.

Newton Form : Differences

For example, given a table of interpolation nodes and corresponding function values:

x	3	1	5	6
$f(x)$	1	-3	2	4

the table of divided differences can be computed according to Table 7.1 as

3	1	2	$-3/8$	$7/40$
1	-3	$5/4$	$3/20$	
5	2	2		
6	4			

Then the interpolation polynomial can be written as

$$p_3(x) = 1 + 2(x-3) - \frac{3}{8}(x-3)(x-1) + \frac{7}{40}(x-3)(x-1)(x-5).$$

Newton Form : Differences

For example, given a table of interpolation nodes and corresponding function values:

x	3	1	5	6
$f(x)$	1	-3	2	4

the table of divided differences can be computed according to Table 7.1 as

3	1	2	$-3/8$	$7/40$
1	-3	$5/4$	$3/20$	
5	2	2		
6	4			

Then the interpolation polynomial can be written as

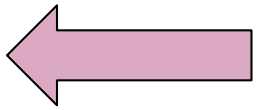
$$p_3(x) = 1 + 2(x-3) - \frac{3}{8}(x-3)(x-1) + \frac{7}{40}(x-3)(x-1)(x-5).$$

Newton Form : Differences

When n is large, calculating the divided difference table can not be done by hand. To derive an efficient algorithm for doing this, introduce the notation $c_{i,j} = f[x_i, x_{i+1}, \ldots, x_{i+j}]$. Then Table 7.1 of divided differences becomes (in the case of $n = 4$):

This actually leads to the following algorithm:

for $(k = 0, 1, \ldots, n)$ {
  $c_{k,0} \leftarrow f(x_k)$;
}
for $(j = 1, 2, \ldots, n)$ {
  for $(i = 0, 1, \ldots, n - j)$ {
    $c_{i,j} \leftarrow (c_{i+1,j-1} - c_{i,j-1})/(x_{i+j} - x_i)$;
  }
}



x_0	$c_{0,0}$	$c_{0,1}$	$c_{0,2}$	$c_{0,3}$	$c_{0,4}$
x_1	$c_{1,0}$	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	
x_2	$c_{2,0}$	$c_{2,1}$	$c_{2,2}$		
x_3	$c_{3,0}$	$c_{3,1}$			
x_4	$c_{4,0}$				

Then $p_n(x)$ can be obtained as

$$p_n(x) = c_{0,0} + c_{0,1}(x - x_0) + c_{0,2}(x - x_0)(x - x_1) + \cdots$$

Newton Form : Differences

Space-efficient:

A closer look at the algorithm reveals that only $c_{0,0}, c_{0,1}, \ldots, c_{0,n}$ are needed in constructing $p_n(x)$, and other $c_{i,j}$ for $i \neq 0$ are just intermediate results. A space-efficient version of the algorithm requires only an array $b = [b_0, b_1, \ldots, b_n]$ of size $n + 1$. First store the given function value $f(x_i)$ into b_i, for $i = 0, 1, \ldots, n$. Note that $b_0 = c_{0,0}$. Then compute divided differences $c_{i,1}$ for $i = 0, 1, \ldots, n - 1$ and store them in $b_1, b_2, \ldots, b_n$. Now $b_0 = c_{0,0}$ and $b_1 = c_{0,1}$ are the first two desired coefficients for $p_n(x)$. Next compute divided differences $c_{i,2}$ for $i = 0, 1, \ldots n - 2$ and store them in $b_2, b_3, \ldots, b_n$. Now $b_0 = c_{0,0}$, $b_1 = c_{0,1}$, and $b_2 = c_{0,2}$. Continue this process until $c_{0,n}$ is computed and stored in b_n. Now the revised and more space-efficient algorithm for calculating the coefficients of $p_n(x)$ reads:

Newton Form : Differences

Space-efficient:

```
for (k = 0, 1, ..., n) {
    b_k ← f(x_k);
}
for (j = 1, 2, ..., n) {
    for (i = n, n − 1, ..., j) {
        b_i ← (b_i − b_{i−1})/(x_i − x_{i−j});
    }
}
```

After this algorithm computes all b_k, the interpolation polynomial $p_n(x)$ can be finally constructed in Newton's form:

$$\begin{aligned} p_n(x) &= b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1) + \cdots \\ &\quad + b_n(x - x_0)(x - x_1)\cdots(x - x_{n-1}). \end{aligned}$$

Newton Form : C++ Code

Code:

```
template<class T>
T newton(const vector<T>& vx, const vector<T>& vy, T x) {
  vector<T> b = vy;
  int n = vx.size() - 1;

  // find coefficients in Newton's form
  for (int j = 1; j <= n; j++)
    for (int i = n; i >= j; i--)
      b[i] = (b[i] - b[i-1])/(vx[i] - vx[i-j]);

  // evaluate interpolation polynomial at x
  T u = b[n];
  for (int i = n - 1; i >= 0; i--) u = b[i] + (x - vx[i])*u;
  return u;
}
```

Newton Form : C++ Code

Conclude:

The advantage of Newton's form is that it is very efficient and the coefficients c_i, $i = 0, 1, \ldots, n$, can be used when later there are more interpolation points available. For example, if later one more point $(x_{n+1}, f(x_{n+1}))$ is given, then $p_{n+1}(x) = p_n(x) + c_{n+1}(x - x_0) \cdots (x - x_n)$ gives $c_{n+1} = (f(x_{n+1}) - p_n(x_{n+1}))/((x_{n+1} - x_0) \cdots (x_{n+1} - x_n))$.

UNIVERSITY OF MANITOBA

Class Inheritance 1

Lecture 3

ECE7650-T13: C++ and Object-Oriented Numeric Computing for Engineers

Date: Marth 14th, 2016

Electrical and Computer Engineering

Before the lecture

http://publib.boulder.ibm.com/iseries/v5r1/ic2924/books/c092712220.htm

The C++ programming language is based on the C language. Although C++ is a descendant of the C language, the two languages are not always compatible.

In C++, you can develop new data types that contain functional descriptions (member functions) as well as data representations. These new data types are called ***classes.*** The work of developing such classes is known as *data abstraction*. You can work with a combination of classes from established class libraries, develop your own classes, or derive new classes from existing classes by adding data descriptions and functions. New classes can contain (***inherit)*** properties from one or more classes. The classes describe the data types and functions available, but they can hide (***encapsulate***) the implementation details from the client programs.

You can define a series of functions with different argument types that all use the same function name. This is called ***function overloading***. A function can have the same name and argument types in base and derived classes.

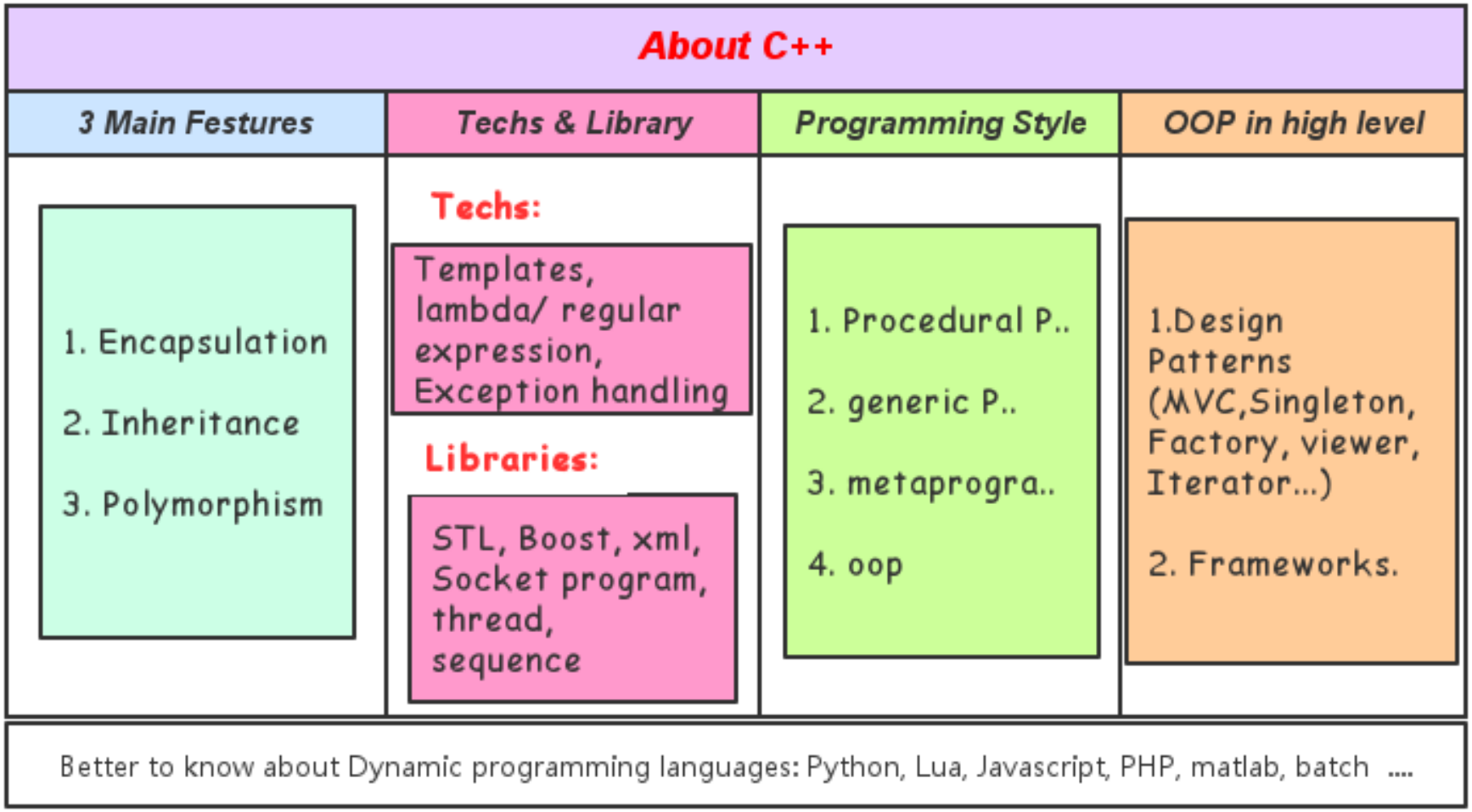
Declaring a class member function in a base class allows you to override its implementation in a derived class. If you use virtual functions, class-dependent behavior may be determined at run time. This ability to select functions at run time, depending on data types, is called ***polymorphism.***

You can redefine the meaning of the basic language operators so that they can perform operations on user-defined classes (new data types), in addition to operations on system-defined data types, such as int, char, and float. Adding properties to operators for new data types is called **operator overloading**.

The C++ language provides **templates** and several keywords not found in the C language. Other features include *try-catch-throw* **exception handling**, stricter type checking and more versatile access to data and functions compared to the C language.

UNIVERSITY

Before the lecture



UNIVERSITY OF MANITOBA

Outline of Class Inheritance

- **Introduction**
- Derived Classes
- Abstract Classes
- Access Control
- Multiple Inheritance
- Run-time Type Information
- Replacing Virtual Functions by Static Polymorphism

UNIVERSITY OF MANITOBA

Introduction

Class Inheritance: A new class can be derived from an existing class.

The new class, called **derived class**, then inherits all members of the existing class, called **base class**, and may provide **additional data, functions, or other members**. This relationship is called inheritance.

The derived class can then behave as a subtype of the base class and an object of the derived class can be assigned to a variable of the base class through **pointers and references**. This is the essential part of what is commonly called **object-oriented programming**.

This chapter deals with various issues in class inheritance. The disadvantage of object-oriented programming is that it may sometimes greatly **affect run-time performance**.
The last section of the chapter presents techniques on what can be done when performance is a big concern.

UNIVERSITY OF MANITOBA

Outline of Class Inheritance

- Introduction
- **Derived Classes**
- Abstract Classes
- Access Control
- Multiple Inheritance
- Run-time Type Information
- Replacing Virtual Functions by Static Polymorphism

UNIVERSITY OF MANITOBA

Derived Classes

- Concept
- Member functions
- Constructor and Destructors
- Coping
- Class Hierarchy
- Virtual Functions
- Virtual Destructors

Derived Classes

Example

Suppose we want to define classes for points in one, two, and three dimensions. A point in 1D can be defined as

```
class Pt {                                  // class for 1D point
private:
  double x;                                 // x coordinate
public:
  Pt(double a = 0) {  x = a; }              // constructor
  void draw() const { cout << x; }          // draw the point
};
```

The function *draw*() is defined to simply print out the coordinate. Since a 2D point has both x and y coordinates, it can be defined to inherit the x coordinate from the class Pt and provide a new member for the y coordinate:

UNIVERSITY OF MANITOBA

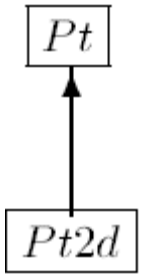
Derived Classes

Example

```
class Pt2d: public Pt {                    // class for 2D point
private:                                   // inherits x from Pt
  double y;                                // y coordinate
public:
  Pt2d(double a = 0, double b = 0): Pt(a), y(b) { }
  void draw() const {                      // draw x coordinate
    Pt::draw();                            // by Pt's draw()
    cout << " " << y;                      // draw y coordinate
  }
```

By putting *:public Pt* after the name of the class *Pt2d* when it is declared, the new class *Pt2d* is defined to be a *derived class* from the *base class Pt*. The derived class *Pt2d* is also called a *subclass* and the base class *Pt* a *superclass*. *Pt2d* has members of class *Pt* in addition to its own members. The derived class *Pt2d* is often said to inherit properties from its base class *Pt*. This relationship is called *inheritance*.

Derived Classes



Now a $Pt2d$ is also a Pt (a two-dimensional point can be regarded as a one-dimensional point by just looking at the x coordinate), and $Pt2d^*$ (pointer to $Pt2d$) can be used as Pt^*. However, a Pt is not necessarily a $Pt2d$ and a Pt^* can not be used as a $Pt2d^*$. The keyword *public* in the definition of $Pt2d$ means that the derived class $Pt2d$ has a public base Pt, which in turn means that a $Pt2d^*$ can be assigned to a variable of type Pt^* by any function without an explicit type conversion. (Inheritance using *private* and *protected* bases is discussed in §8.3.) The opposite conversion, from a pointer to base class to a pointer to derived class, must be explicit.

Derived Classes: type conversion

Example

```
void f(Pt p1, Pt2d p2) {
  Pt* q1 = &p2;                    // OK. Every Pt2d is a Pt
  Pt2d* q2 = &p1;                  // error, not every Pt is a Pt2d
}

void g(Pt p1, Pt2d p2) {           // statements below are legal
  Pt* q1 = &p2;                    // Every Pt2d is a Pt
  Pt2d* q2 = static_cast<Pt2d*>(q1);
                                   // explicit type conversion

  q2->draw();                      // x, y of p2 are printed

  Pt2d* r2 = static_cast<Pt2d*>(&p1);
                                   // explicit type conversion

  r2->draw();                      // OK, but y may be garbage
}
```

Derived Classes: type conversion

Example

```
int main() {                        // test what is printed out

  Pt a(5);                          // create a 1D point
  Pt2d b(4, 9);                     // create a 2D point
  g(a, b);                          // call g() on a and b

}
```

During the second draw in calling $g()$, the 1D point a does not have a y-coordinate and thus some garbage should be printed out. The first draw in $g()$ on the 2D point b should print out both coordinates of b correctly.

Derived Classes: type conversion

Comments:

An object of a derived class can be treated as an object of its base class when manipulated through pointers and references. The opposite must be through explicit type conversion using *static_cast* (at compile-time) or *dynamic_cast* (at run-time; see §8.5), although the result of such conversions can not be guaranteed in general. For example, the second cast in the function $g()$ above tries to convert a pointer to a base class to a pointer to a derived class and may result in an object with an undefined y coordinate.

The operator *static_cast* converts between related types such as from one pointer type to another or from an enumeration to an integral type. There is another cast, called *reinterpret_cast*, which converts between unrelated types such as an integer to a pointer. These casts may not be portable, can be even dangerous, but are sometimes necessary. They should be avoided when possible.

UNIVERSITY OF MANITOBA

Derived Classes

Points:

- Base, derived class(superclass, subclass)
- Syntax, format
- Figure, single inheritance
- (Class) Type conversion

8.1.1 Member functions

1 invisible members

Member functions of a derived class can not access the private part of a base class, although a derived class contains all members of the base class. For example, the function $Pt2d::draw()$ can not be defined as

```
void Pt2d::draw() const { // incorrect
  cout << x << " " << y;  // x is not accessible
}
```

A member of a derived class that is inherited from the private part of a base class may be called an *invisible member* of the derived class. Thus an invisible member of a class can not be accessed by members or friends even of the same class. This is for information hiding, since otherwise a private member of a class could be accessed freely just by deriving a new class from it.

8.1.1 Member functions

2 access the invisible member

However, the hidden member x of the derived class $Pt2d$ is accessible through the public member function $Pt::draw()$:

```
void Pt2d::draw() const { // draw the point
  Pt::draw();             // draw x coordinate by Pt's draw()
  cout << " " << y;       // draw y coordinate
}
```

Note the class qualifier $Pt::$ must be used to refer to a member function of class Pt.

(private) Data -> set, (public) get functions

8.1.2 Constructor and Destructors

1 Syntax

In defining the constructor of a derived class, the constructor of the base class must be called in the initializer list (like class object members; see §5.6) if the base class has a constructor. For example,

```
Pt2d::Pt2d(double a = 0, double b = 0): Pt(a), y(b) { }
```

It can also be defined alternatively as

```
Pt2d::Pt2d(double a = 0, double b = 0): Pt(a) { y = b; }
```

But it can not be defined as

```
Pt2d::Pt2d(double a = 0, double b = 0) { Pt(a); y = b; }
```

The third definition is wrong since the construction of base class *Pt* is not put in the initializer list.

8.1.2 Constructor and Destructors

2 Calling order

Construction: derived class constructor -> members -> derived class itself
Destruction: derived class destructor -> members -> the base class destructor.

A derived class constructor can specify initializers for its own members and immediate bases only. The construction of an object of a derived class starts from the base class, then the members, and then the derived class itself. Its destruction is in the opposite order: first the derived class itself, then its members, and then the base class. Members and bases are constructed in order of declarations in the derived class and destructed in the reverse order.

8.1.2 Constructor and Destructors

```
class Pt
{
private:
    double x;
public:
    Pt(double a=0){
        x=a;
        cout<<"Pt --- Base class Constructor!\n";
    }
    ~Pt(){
        cout<<"~Pt --- Base class Destructor!\n";
    }
};

class Pt2d : public Pt
{
private:
    double y;
public:
    Pt2d(double a = 0, double b =0): Pt(a){
        this->y = b;
        cout<<"Pt2d --- Derived class Constructor!\n";
    }
    ~Pt2d(){
        cout<<"~Pt2d --- Derived class Destructor!\n";
    }
};
```

```
int main()
{
    Pt2d myPt2d(1,2);
    // system pause
    cin.get();
    return 0;
}
```

```
Pt --- Base class Constructor!
Pt2d --- Derived class Constructor!

~Pt2d --- Derived class Destructor!
~Pt --- Base class Destructor!
```

8.1.2 Constructor and Destructors

3 Tips about destructors

In the definition of the destructor of a derived class, only spaces allocated in the derived class need be explicitly deallocated; space allocated in the base class are freed implicitly by the destructor of the base class.

```
// Safe delete pointers
template<class T>
inline void SafeDelete(T*& ptr)
{
    if(ptr)
    {
        delete ptr;
        ptr = nullptr;
    }
}
// Safe delete array
template<class T>
inline void SafeDeleteArray(T*& pArray)
{
    if(pArray)
    {
        delete[] pArray;
        pArray = nullptr;
    }
}
```

8.1.3 Coping

Copying of class objects is defined by the copy constructor and assignment. For example,

```
class Pt {
public:           // ... in addition to other members
  Pt(const Pt& p) { x = p.x; }   // copy constructor
  Pt& operator=(const Pt& p) {   // copy assignment
    if (this != &p) x = p.x;
    return *this;
  }
};

Pt2d p2;
Pt p1 = p2;                  // construct p1 from Pt part of p2
p1 = p2;                     // assign Pt part of p2 to p1
```

Since the copy functions of *Pt* do not know anything about *Pt2d*, only the *Pt* part (x-coordinate) of a *Pt2d* object is copied and other parts are lost. This is commonly called *slicing*. It can be avoided by passing pointers and references; see §8.1.5.

8.1.4 Class Hierarchy

Key 1: A derived class can be a base class of another derived class.

Example 1

```
class Pt3d: public Pt2d {      // point in 3D
private:                       // inherits x,y from Pt2d
  double z;                    // z coordinate
public:
  Pt3d(double a = 0, double b = 0, double c = 0)
    : Pt2d(a, b), z(c) { }
  void draw() const {          // draw the point
    Pt2d::draw();              // draw x,y by Pt2d's draw()
    cout << " " << z;          // draw z coordinate
  }
};
```

8.1.4 Class Hierarchy

Key 2: A derived class can inherit from two or more base classes. This is called **Multiple inheritance.**

```
class Circle_in_Triangle: public Circle, public Triangle {
  // ...                    // two bases
public:
  void draw() {              // override Circle::draw()
                             //   and Triangle::draw()
    Circle::draw();          // call draw() of Circle
    Triangle::draw();        // call draw() of Triangle
  }
  void add_border();         // override Circle::add_border()
};
```

A set of related classes derived through inheritance is called a class hierarchy.

8.1.5 Virtual Functions

Virtual functions can be declared in a base class and may be redefined (also called overridden) in each derived class when necessary. They have the same name and same set of argument types in both base class and derived class, but perform different operations. When they are called, the system can guarantee the correct function be invoked according to the type of the object at run-time. For example, the class *Pt* can be redefined to contain a virtual function:

```
class Pt {
private:
  double x;
public:
  Pt(double a = 0) {  x = a; }
  virtual void draw() const { cout << x; }   // virtual fcn
};
```

UNIVERSITY OF MANITOBA

8.1.5 Virtual Functions

The member $Pt::draw()$ is declared to be a virtual function. The declarations of $Pt2d$ and $Pt3d$ remain unchanged. The keyword *virtual* indicates that $draw()$ can act as an interface to the $draw()$ function defined in this class and the $draw()$ functions defined in its derived classes $Pt2d$ and $Pt3d$. When $draw()$ is called, the compiler will generate code that chooses the right $draw()$ for a given Pt object ($Pt2d$ and $Pt3d$ objects are also Pt objects). For example, if a Pt object is actually a $Pt3d$ object, the function $Pt3d::draw()$ will be called automatically by the system.

A virtual function can be used even if no class is derived from the class, and a derived class need not redefine it if the base class version of the virtual function works fine. Now functions can be defined to print out a set of points (some may be 1D or 2D while others may be 3D):

UNIVERSITY OF MANITOBA

8.1.5 Virtual Functions

Example

```
class Pt3d: public Pt2d
{
private:
    double z;
public:
    Pt3d(double a =0, double b=0, double c=0)
        : Pt2d(a,b)
        , z(c){}
    ~Pt3d(){}
    void showInfo(){
        Pt2d::showInfo();
        cout<<"z = "<<this->z<<";";
    }
    void draw(){
        this->showInfo();
    }
};
```

```
class Pt
{
private:
    double x;
public:
    Pt(double a=0): x(a){}
    ~Pt(){}
    void showInfo(){
        cout<<"x = "<<this->x<<"; ";
    }
    virtual void draw(){
        showInfo();
    }
};
class Pt2d : public Pt
{
private:
    double y;
public:
    Pt2d(double a = 0, double b =0): Pt(a),y(b){}
    ~Pt2d(){}
    void showInfo(){
        Pt::showInfo();
        cout<<"y = "<<this->y<<";";
    }
    void draw(){
        this->showInfo();
    }
};
```

8.1.5 Virtual Functions

```
  cout<<"===== test 1 ========\n";
  Pt2d myPt2d(1,2);
  Pt myPt = myPt2d;
  myPt.showInfo();

  cout<<"\n\n===== test 2 ========\n";
  Pt2d * myPointPt2d = new Pt2d(1,2);
  Pt * myPointPt = myPointPt2d;
  myPointPt->showInfo();
```

```
cout<<"\n\n===== test 3 ========\n";
Pt a(5);
Pt2d b(4,9);
Pt3d c(7,7,7);
vector<Pt *> v(3);
v[0] = &a; v[1] =&b; v[2] =&c;
for (int i = 0; i <v.size(); i++)
{
    v[i]->showInfo();
    cout<<endl;
}
cout<<"\n\n===== test 4 ========\n";
for (int i = 0; i <v.size(); i++)
{
    v[i]->draw();
    cout<<endl;
}
```

```
===== test 1 ========
x = 1;

===== test 2 ========
x = 1;

===== test 3 ========
x = 5;
x = 4;
x = 7;


===== test 4 ========
x = 5;
x = 4; y = 9;
x = 7; y = 7;z = 7;
```

8.1.5 Virtual Functions

For library <*vector*> see §10.1.1. This will work even if the function $h()$ was written and compiled before the derived classes $Pt2d$ and $Pt3d$ were even conceived of, and the points (1D, 2D, or 3D) in the argument of $h()$ can be generated dynamically at run-time by another function. Since $draw()$ is declared to be *virtual* in the base class, the system guarantees that in the call $v[i]{->}draw()$, the function $Pt::draw()$ is invoked if $v[i]$ is a pointer to Pt, $Pt2d::draw()$ is invoked if $v[i]$ is a pointer to $Pt2d$, and $Pt3d::draw()$ is invoked if $v[i]$ is a pointer to $Pt3d$. This is a key aspect of class inheritance. When used properly, it is a cornerstone of object-oriented design.

8.1.5 Virtual Functions: polymorphism

Polymorphism:

Dynamic:

Getting the right correspondence between the function (e.g., among different versions of *draw*()) and the type (e.g., among *Pt*, *Pt2d*, or *Pt3d*) of an object is called run-time *polymorphism*. A type with virtual functions is called a *polymorphic type*. Run-time polymorphism is also called *dynamic binding*. To get polymorphic behavior, member functions called must be *virtual* and objects must be manipulated through pointers or references.

8.1.5 Virtual Functions: polymorphism

Polymorphism:

Static:

When manipulating objects directly (rather than through pointers or references), their exact types must be known at compile-time, for which run-time polymorphism is not needed and templates may be used. See §8.6 for examples that can be implemented using either templates or virtual functions. In contrast, what templates provide is often called compile-time polymorphism, or *static polymorphism*.

8.1.5 Virtual Functions: polymorphism

Comments:

When run-time polymorphism is not needed, the scope resolution operator :: should be used, as in $Pt::draw()$ and $Pt2d::draw()$. When a virtual function is inline, function calls that do not need run-time polymorphism can be specified by using the scope resolution operator :: so that inline substitution may be made. This is reflected in the definitions of $Pt2d::draw()$ and $Pt3d::draw()$.

Polymorphic behavior can be achieved through pointers to virtual member functions.

UNIVERSITY OF MANITOBA

8.1.6 Virtual Destructors

Example 1

```
class B {
  double* pd;
public:
  B() {                                   // constructor of B
    pd = new double [20];
    cout << "20 doubles allocated\n";
  }
  ~B() {                                  // destructor of B
    delete[] pd;
    cout << "20 doubles deleted\n";
  }
};

class D: public B {                       // derive D from B
  int* pi;
public:
  D(): B() {                              // constructor of D
    pi = new int [1000];
    cout << "1000 ints allocated\n";
  }
  ~D() {                                  // destructor of D
    delete[] pi;
    cout << "1000 ints deleted\n";
  }
};
```

UNIVERSITY OF MANITOBA

8.1.6 Virtual Destructors

Example 1

Then the code

```
int main() {
  B* p = new D;
     // 'new' constructs a D object
  delete p;
     // 'delete' frees a B object since p is a pointer to B
}
```

will produce the output

```
20 doubles allocated
1000 ints allocated
20 doubles deleted
```

Proper cleanup is not achieved here. The reason is that the *new* operator constructed an object of type D (allocated 20 doubles and 1000 ints when D's constructor was called), but the *delete* operator cleaned up an object of type B pointed to by p (freed 20 doubles when B's destructor was implicitly called). In other words, run-time polymorphism did not apply to the destructor of class D. If it were applied, the system would have detected the type of the object pointed to by p and called the destructor of the correct type (class D).

8.1.6 Virtual Destructors

Solution

This problem can be easily solved by declaring the destructor of the base class B to be *virtual* :

```
class B {
  double* pd;
public:
  B() {
    pd = new double [20];
    cout << "20 doubles allocated\n";
  }
  virtual ~B() {                          // a virtual destructor
    delete[] pd;
    cout << "20 doubles deleted\n";
  }
};
```

The definition of the derived class D is unchanged.

8.1.6 Virtual Destructors

Solution

```
20 doubles allocated
1000 ints allocated
1000 ints deleted
20 doubles deleted
```

From this, the orders of construction and destruction of objects of derived and base classes can also be clearly seen; see §8.1.2.

In general, a class with a virtual function should have a virtual destructor, because run-time polymorphism is expected for such a class. However, other base classes such as class B above may also need a virtual destructor for proper cleanups.

UNIVERSITY OF MANITOBA

Class Inheritance 2

Lecture 4

ECE7650-T13: C++ and Object-Oriented Numeric Computing for Engineers

Date: Marth 11th, 2016

Electrical and Computer Engineering

UNIVERSITY OF MANITOBA

Outline of Class Inheritance

- Introduction
- Derived Classes
- **Abstract Classes**
- Access Control
- Multiple Inheritance
- Run-time Type Information
- Replacing Virtual Functions by Static Polymorphism

Abstract Classes

Pure function:

Some virtual functions in a base class can only be declared but can not be defined, since the base class may not have enough information to do so and such virtual functions are only meant to provide a common interface for the derived classes. A virtual function is called a *pure virtual function* if it is declared but its definition is not provided. The initializer "= 0" makes a virtual function a pure one. For example,

```
class Shape {
public:
  virtual void draw() = 0;         // pure virtual function
  virtual void rotate(int i) = 0; // rotate i degrees
  virtual bool is_closed() = 0;    // pure virtual function
  virtual ~Shape() { }             // empty virtual destructor
};
```

Abstract Classes

Pure function:

The member functions *rotate*(), *draw*(), and *is_closed*() are pure virtual functions recognized by the initializer "= 0". Note the destructor ~*Shape*() is not a pure virtual function since it is defined, although it does nothing. This is an example of a function that does nothing but is useful and necessary. The class *Shape* is meant to be a base class from which other classes such as *Triangle* and *Circle* can be derived. Its member functions *rotate*(), *draw*(), and *is_closed*() can not be defined since there is not enough information to do so.

Abstract Classes

Abstract class:

A class with one or more pure virtual functions is called an *abstract class*. No objects of an abstract class can be created. For example, it is illegal to define:

```
Shape s;    // object of an abstract class can not be defined
```

An abstract class can only be used as an interface and as a base for derived classes. For example, a special shape: *Circle* can be derived from it as

```
class Circle: public Shape {    // derived class for circles
private:
  Pt2d center;                  // center of circle
  double radius;                // radius of circle
public:
  Circle(Pt2d, double);         // constructor
  void rotate(int) { }          // override Shape::rotate()
  void draw();                  // override Shape::draw()
  bool is_closed() { return true; }   // a circle is closed
};
```

Abstract Classes

Abstract class:

The member functions of *Circle* can be defined since there is enough information to do so. These functions override the corresponding functions of *Shape*. The function *Circle* :: *rotate*() does nothing since a circle does not change when being rotated, and *Circle* :: *is_closed*() returns *true* since a circle is a closed curve. *Circle* is not an abstract class since it does not contain pure virtual functions, and consequently objects of *Circle* can be created.

We need to override all the pure function in the derived class in order to make it effective.

Abstract Classes

Abstract class:

A pure virtual function that is not defined in a derived class remains a pure virtual function. Such a derived class is also an abstract class. For example,

```
class Polygon: public Shape {    // abstract class
public:
  bool is_closed() { return true; }

        // override Shape::is_closed()
        // but draw() & rotate() still remain undefined.
 };
```

UNIVERSITY OF MANITOBA

Abstract Classes

Abstract class:

```
class Triangle: public Polygon { // not abstract any more
private:
  Pt2d* vertices;
public:
  Triangle(Pt2d*);
  ~Triangle() { delete[] vertices; }
  void rotate(int);                    // override Shape::rotate()
  void draw();                         // override Shape::draw()
};
```

Class *Polygon* remains an abstract class since it has pure virtual functions *draw*() and *rotate*(), which are inherited from its base class *Shape* but have not been defined. Thus objects of *Polygon* can not be created. However, *Triangle* is no longer an abstract class since it does not contain any pure virtual functions (definitions of its members *rotate*() and *draw*() are omitted here) and objects of *Triangle* can be created.

Abstract Classes

Conclusion:

- Single inheritance. (subclass→superclass)
- Class hierarchy
- Dynamic Polymorphism (virtual function)
- Abstract Class (pure function)

UNIVERSITY OF MANITOBA

Abstract Classes

A problem:

Below is a design problem for iterative numeric methods on solving linear systems of algebraic equations $Ax = b$, where A is a square matrix, b is the right-hand side vector, and x is the unknown vector. In §6.6 the conjugate gradient (CG) method is implemented for a Hermitian and positive definite matrix A and in §11.3 the generalized minimum residual (GMRES) method is introduced for any nonsingular matrix A. These two iterative methods require only matrix-vector multiplication and some vector operations in order to solve the linear system $Ax = b$. Details of CG and GMRES are not needed here. Instead, a class hierarchy is designed to apply CG and GMRES to full, band, and sparse matrices A. In some applications, most entries of the matrix A are not zero and all entries of A are stored. Such a matrix storage format is called a *full matrix*. In some other applications, the entries of A are zero outside a band along the main diagonal of A. Only entries within the band (zero or nonzero) may be stored to save memory and such a format is called a *band matrix*. Yet in other applications most entries of A are zero and only nonzero entries are stored to further save memory; such a format is called a *sparse matrix*. Three different classes need be defined for these three matrix storage formats, of which the full matrix format is given in §6.3 where every entry of a matrix is stored.

Abstract Classes

A problem:

1. Question: solve the Ax = b, where A is a square matrix.
2. Method: Conjugate-gradient (CG)
3. Operations: matrix-vector multiplication, vector operations
4. Challenge: 3 different matrices.(full, band, sparse Matrix)
5. Goal: design. (class hierarchy.) Details are not needed here.

Work fine→ efficient → extendibility, maintainability, readability (elegant)

Abstract Classes

Analysis:

The objective of such a design is that the CG and GMRES methods are defined only once, but are good for all three matrix storage formats, instead of providing one definition for each storage format. This should also be extendible possibly by other users later to other matrix formats such as symmetric sparse, band, and full matrices (only half of the entries need be stored for symmetric matrices to save memory). Since CG and GMRES depend on a matrix-vector product, which must be done differently for each matrix storage format, the correct matrix-vector multiplication function has to be called for a particular matrix format. To provide only one definition for CG and GMRES, they can be defined for a base class and inherited for derived classes representing different matrix formats. To ensure the correct binding of matrix-vector multiplication to each matrix storage format, a virtual function can be utilized.

Abstract Classes

Implementation:

```
class AbsMatrix {                    // base class
public:
  virtual ~AbsMatrix() { }           // a virtual destructor

  virtual Vtr operator*(const Vtr &) const = 0;
                                     //  matrix vector multiply
  int CG();
  int GMRES();
};

int AbsMatrix::CG() { /* ... definition omitted here */ }
int AbsMatrix::GMRES() { /* ... definition omitted here */ }
```

Abstract Classes

Implementation:

This matrix class serves as an interface for all derived classes.

1. Since there is not enough information to define the matrix-vector multiplication operator *, it is declared as a pure virtual function.
2. For such a class, it is natural to define a virtual destructor to ensure that objects of derived classes are properly cleaned up, although this virtual destructor does nothing here.
3. This class AbsMatrix is actually an abstract class so that no objects of AbsMatrix can be created.
4. The functions CG() and GMRES() can be fully defined, which can be inherited and used in derived classes once the matrix-vector multiply operator * is defined.

Then, let's see how the derived classes are defined:

Abstract Classes

Implementation:

```
class FullMatrix: public AbsMatrix {
public:     // ... in addition to other members
  Vtr operator*(const Vtr&) const;
            // multiply a full matrix with a vector
};

class SparseMatrix: public AbsMatrix {
public:     // ... in addition to other members
  Vtr operator*(const Vtr&) const;
            // multiply a sparse matrix with a vector
};

class BandMatrix: public AbsMatrix {
public:     // ... in addition to other members
  Vtr operator*(const Vtr&) const;
            // multiply a band matrix with a vector
};
```

UNIVERSITY OF MANITOBA

Abstract Classes

Implementation:

The matrix-vector multiply operator $*$ can be defined (but omitted here) for each of the derived classes and thus $CG()$ and $GMRES()$ can be invoked for objects of $FullMatrix$, $BandMatrix$, and $SparseMatrix$. For example,

```
void f(FullMatrix& fm, BandMatrix& bm, SparseMatrix& sm) {
  fm.CG();          // call CG() on FullMatrix fm
  bm.CG();          // call CG() on BandMatrix bm
  sm.CG();          // call CG() on SparseMatrix sm
}

void g(vector<AbsMatrix*>& vm) {
  for (int i = 0; i < vm.size(); i++) {
    vm[i]->GMRES();
    // the object pointed to by vm[i] could be FullMatrix,
    // BandMatrix, or SparseMatrix. Its exact type may not
    // be known at compile-time.
  }
}
```

UNIVERSITY OF MANITOBA

Outline of Class Inheritance

- Introduction
- Derived Classes
- Abstract Classes
- **Access Control**
- Multiple Inheritance
- Run-time Type Information
- Replacing Virtual Functions by Static Polymorphism

UNIVERSITY OF MANITOBA

Access Control

- **Access to members**
- **Access to Base Classes**

UNIVERSITY OF MANITOBA

Access Control 1: access to Members

A member (it can be a function, type, constant, etc.. as well as a data member) of a class can be declared to be private, protected, or public.

- A *private* member can be used only by members and friends of the class in which it is declared,
- A *protected* member can be used by members and friends of the class in which it is declared. Furthermore, it becomes a private, protected, or public member of classes derived from this class; as a member of a derived class, it can be used by members and friends of the derived class and maybe by other members in the class hierarchy depending on the form of inheritance. See §8.3.2 for more details.
- A *public* member can be used freely in the program. It also becomes a private, protected, or public member of classes derived from this class, depending on the form of inheritance. See §8.3.2 for more details.

UNIVERSITY OF MANITOBA

Access Control 1: access to Members

Example 1

```
class B {
private:
  int i;                          // a private member
protected:
  float f;                        // a protected member
public:
  double d;                       // a public member
  void g1(B& b) { f = b.f; }      // f and b.f are accessible
};

void g() {
  B bb;                           // by default constructor
  bb.i = 5;                       // error, can not access bb.i
  bb.f = 3.14;                    // error, can not access bb.f
  bb.d = 2.71;                    // bb.d is accessible freely
}
```

The specifiers protected and private mean the same when inheritance is not involved.

Access Control 1: access to Members

Comments on "Protected":

Protected members of a class are designed for use by derived classes and are not intended for general use. They provide another layer of information hiding, similar to private members. The protected part of a class usually provides operations for use in the derived classes, and normally does not contain data members since it can be more easily accessed or abused than the private part. For this reason the data member x in the class Pt in §8.1 is better kept *private* than *protected*. Despite this, the derived classes $Pt2d$ and $Pt3d$ can still print out the value of x.

UNIVERSITY OF MANITOBA

Access Control 2: access to Base Classes

A base class can be declared *private*, *protected*, or *public*, in defining a derived class. For example,

```
class B { /* ... */ };                  // a class
class X: public B { /* ... */ };        // B is a public base
class Y: protected B { /* ... */ };     // B is a protected base
class Z: private B { /* ... */ };       // B is a private base
```

Public derivation makes the derived class a subtype of its base class and is the most common form of derivation. Protected and private derivations are used to represent implementation details. No matter what form the derivation is, a derived class contains all the members of a base class (some of which may have been overridden), although members inherited from the private part of the base class are not directly accessible even by members and friends of the derived class (they are called *invisible members* of the derived class in §8.1.1).

Access Control 2: access to Base Classes

Rules:

The form of derivation controls access to the derived class's members inherited from the base class and controls conversion of pointers and references from the derived class to the base class. They are defined as follows. Suppose that class D is derived from class B.

- If B is a private base, its public and protected members become private members of D. Only members and friends of D can convert a D^* to a B^*.
- If B is a protected base, its public and protected members become protected members of D. Only members and friends of D and members and friends of classes derived from D can convert a D^* to a B^*.
- If B is a public base, its public members become public members of D and its protected members become protected members of D. Any function can convert a D^* to a B^*. In this case, D is called a *subtype* of B.

Access Control 2: access to Base Classes

Example 1:

Since X has a public base B, it has three public members: d and $g1()$ (inherited from B), and $g2()$ (its own), two protected members: f (inherited from B) and s (its own), but no private members. X also inherits a member i from base B, but this member is invisible in X. The protected member f of class B and member f of the object b of type B are accessible in the member function $B::g1(B\ b)$. Similarly, the protected (inherited from base class) member f of class X and member f of the object b of type X are accessible in the member function $X::g2(X\ b)$.

```
class B {
private:
  int i;
protected:
  float f;
public:
  double d;
  void g1(B b) { f = b.f; }       // f and b.f are accessible
};                                // by a member of B

class X: public B {               // public derivation
protected:
  short s;
public:
  void g2(X b) { f = b.f; }      // f and b.f are accessible
};                                // by members of X
```

UNIVERSITY OF MANITOBA

Access Control 2: access to Base Classes

Example 2:

```
class X: public B {                    // public derivation
protected:
  short s;
public:
  void g2(X b) { f = b.f; }            // f and b.f are accessible
  void g3(B b) { f = b.f; }            // error: b.f not accessible
};
```

In the definition of $X::g3(B\ b)$, the protected member f of object b of type B is not accessible. Member $b.f$ of object b of type B is only accessible by members and friends of class B. Note the only difference between $X::g2(X)$ and $X::g3(B)$ is the type of their arguments.

Form

Access Control 2: access to Base Classes

Example 2:

A derived class inherits members (e.g., $B::f$ above) that are *protected* and *public* in its base class. These inherited members (e.g., $X::f$) of the derived class are accessible by members and friends of the derived class. However, the derived class can not access protected members (e.g., $b.f$ in the definition of $X::g3(B)$) of an object of the base class, just as it can not access protected members of objects of any other class.

No matter what type of inheritance it is, a private member of a base class can not be used in derived classes to achieve a strong sense of information hiding, since otherwise a private member of any class could be accessed and possibly abused easily by defining a derived class from it.

UNIVERSITY OF MANITOBA

Access Control 2: access to Base Classes

Example 3:

Access to an inherited member may be adjusted to that of the base class by declaring it *public* or *protected* in the derived class using the scope resolution operator :: as in the following example.

```
class B {
private:
  int i;
protected:
  float f;
public:
  double d;
};

class Y: protected B {      // protected derivation
protected:
  short s;
public:
  B::d;                     // B::d is adjusted to be public
};
```

UNIVERSITY OF MANITOBA

Access Control 2: access to Base Classes

Example 4:

If a derived class adds a member with the same name as a member of a base class, the new member in the derived class hides the inherited member from the base class. The inherited member may be accessible using the scope resolution operator preceded by the base class name. For example,

```
class B {
public:
  double d;
};

class Z: private B {                    // private derivation
private:
  double d;                             // this d hides B::d
public:
  void f(Z& z) {
    z.d = 3.1415926;                    // Z::d instead of B::d
    z.B::d = 2.718;                     // z's member B::d
  }
};
```

UNIVERSITY OF MANITOBA

Access Control 2: access to Base Classes

Example 5:

The derivation specifier for a base class can be left out. In this case, the base defaults to a private base for a class and a public base for a struct:

```
class  XX: B { /* ... */ };          // B is a private base
struct YY: B { /* ... */ };          // B is a public base
```

Access Control 2: access to Base Classes

Conclusion:

Base Class	Inheritance	Derived Class
Private	Public	No access
protected		Protected
public		Public
Private	Protected	No access
protected		Protected
public		Protected
Private	Private	No access
protected		Private
public		Private

UNIVERSITY OF MANITOBA

Class Inheritance 3

Lecture 5

ECE7650-T13: C++ and Object-Oriented Numeric Computing for Engineers

Date: Marth 11th, 2016

Electrical and Computer Engineering

UNIVERSITY OF MANITOBA

Outline of Class Inheritance

- Introduction
- Derived Classes
- Abstract Classes
- Access Control
- **Multiple Inheritance**
- Run-time Type Information
- Replacing Virtual Functions by Static Polymorphism

UNIVERSITY OF MANITOBA

Multiple Inheritance

- **Introduction**
- Ambiguity Resolution
- Replicated Base Classes
- Virtual Base Classes
- Access Control in Multiple Inheritance

Multiple Inheritance: Introduction

A class can be directly derived from two or more base classes. This is called ***multiple inheritance***. In contrast, derivation from only one direct base class is called ***single inheritance.*** This section talks about multiple inheritance.

Example:

```
class Circle {
  // ...
public:
  virtual void draw();              // draw the circle
  virtual void add_border();        // add border to circle
  double area();                    // find its area
  void dilate(double d);            // enlarge circle d times
};
```

UNIVERSITY OF MANITOBA

Multiple Inheritance: Introduction

Example:

```
class Triangle {
  // ...
public:
  virtual void draw();              // draw the triangle
  double area();                    // find its area
  void refine();                    // refine into 4 triangles
};
class Circle_in_Triangle: public Circle, public Triangle {
  // ...                            // two bases
public:
  void draw() {                     // override Circle::draw()
                                    //   and Triangle::draw()
    Circle::draw();                 // call draw() of Circle
    Triangle::draw();               // call draw() of Triangle
  }
  void add_border();                // override Circle::add_border()
};
```

The derived class ***Circle_in_Triangle*** inherits properties from both Circle and Triangle.

UNIVERSITY OF MANITOBA

Multiple Inheritance: Introduction

Example:

```
void f(Circle_in_Triangle& ct) {
  ct.dilate(5.0);          // call Circle::dilate()
  ct.refine();             // call Triangle::refine()
  ct.draw();               // call Circle_in_Triangle::draw()
}

double curvature(Circle*);          // curvature of a curve
vector<double> angles(Triangle*);   // angles of a triangle

void g(Circle_in_Triangle* pct) {
  double c = curvature(pct);
  vector<double> a = angles(pct);
}
```

Since public derivation is used for both base classes, any function can convert a $Circle\_in\_Triangle^*$ to $Circle^*$ or $Triangle^*$.

UNIVERSITY OF MANITOBA

Multiple Inheritance

- Introduction
- **Ambiguity Resolution**
- Replicated Base Classes
- Virtual Base Classes
- Access Control in Multiple Inheritance

8.4.1 Ambiguity Resolution

When two base classes have members with the same name, they can be resolved by using the scope resolution operator. For example, both *Circle* and *Triangle* have a function named *area*(). They must be referred to with the class names:

```
void h(Circle_in_Triangle* pct) {
  double ac = pct->Circle::area();        // OK
  double at = pct->Triangle::area();      // OK
  double aa = pct->area();                // ambiguous, error
  pct->draw();                            // OK
}
```

Overload resolution is not applied across different types. In particular, ambiguities between functions from different base classes are not resolved based on argument types.

8.4.1 Ambiguity Resolution

A using-declaration can bring different functions from base classes to a derived class and then overload resolution can be applied:

```
class A {
public:
  int g(int);
  float g(float);
};

class B {
public:
  char g(char);
  long g(long);
};

class C: public A, public B {
public:
  using A::g;                    // bring g() from A
  using B::g;                    // bring g() from B
  char g(char);                  // it hides B::g(char)
  C g(C);
};

void h(C& c) {
  c.g(c);                        // C::g(C) is called
  c.g(1);                        // A::g(int) is called
  c.g(1L);                       // B::g(long) is called
  c.g('E');                      // C::g(char) is called
  c.g(2.0);                      // A::g(float) is called
}
```

UNIVERSITY OF MANITOBA

8.4.2 Replicated Base Classes

Problems:

With the possibility of derivation from two bases, a class can be a base twice. For instance, if both ***Circle*** and ***Triangle*** are derived from a class ***Shape***, then the base class ***Shape*** will be inherited twice in the class ***Circle_in_Triangle***.

Example:

```
class Shape {
protected:
  int color;                        // color of shape
public:
  // ...
  virtual void draw() = 0;          // a pure virtual function
};
```

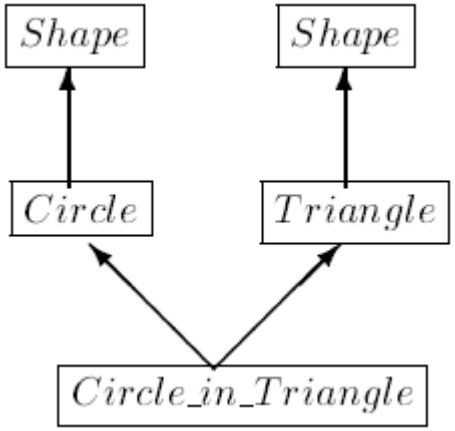
8.4.2 Replicated Base Classes

Example:

```
class Circle: public Shape {
  // ...                              // inherit color for Circle
public:
  void draw();                        // draw circle
};
class Triangle: public Shape {
  // ...                              // color for Triangle
public:
  void draw();                        // draw the triangle
};
class Circle_in_Triangle: public Circle, public Triangle {
  // ...                              // two bases
public:
  void draw();                        // override Circle::draw()

};                                    //   and Triangle::draw()
```

8.4.2 Replicated Base Classes

This causes no problems if the colors of a *Circle* and a *Triangle* for an object of *Circle_in_Triangle* can be different. Indeed, two copies of *Shape* are needed to store the colors. This class hierarchy can be represented as



To refer to members of a replicated base class, the scope resolution operator must be used. For example,

```
void Circle_in_Triangle::draw() {
  int cc = Circle::color;          // or Circle::Shape::color
  int ct = Triangle::color;        // or Triangle::Shape::color
  // ...
  Circle::draw();
  Triangle::draw();
}
```

A virtual function of a replicated base class can be overridden by a single function in a derived class. For example, *Circle_in_Triangle*::*draw*() overrides *Shape*::*draw*() from the two copies of *Shape*.

8.4.3 Virtual Base Classes

Question:

Often a base class need not be replicated. That is, only one copy of a replicated class need be inherited for a derived class object.

Solution:

This can be done by specifying the base to be virtual. Every virtual base of a derived class is represented by the same (shared) object. ---- (C11→ virtual inheritance.)

Example:

If the circle and triangle in Circle_in_Triangle must have the same color, then only one copy of shape is needed for storing the color information. The base class Shape then should be declared to be virtual.

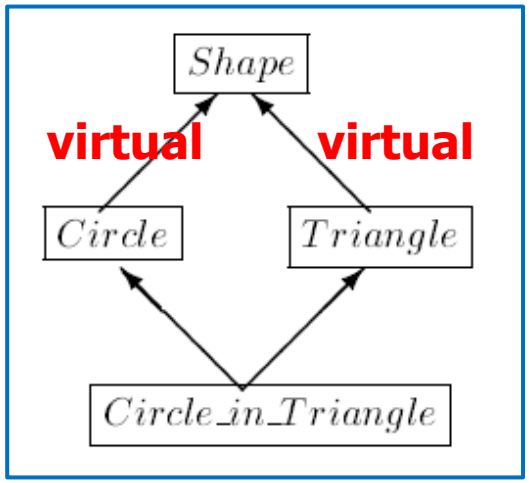
UNIVERSITY OF MANITOBA

8.4.3 Virtual Base Classes

```
class Shape {
  int color;                          // color of shape
public:
  // ...
  virtual void draw() = 0;            // pure virtual function
};

class Circle: public virtual Shape { // virtual base Shape
  // ...                             // inherit color
public:                              //   for Circle
  void draw();                       // draw circle
};

class Triangle: public virtual Shape { // virtual base Shape
  // ...                               // inherit color
public:                                //   for Triangle
  void draw();                         // draw triangle
};

class Circle_in_Triangle: public Circle, public Triangle {
  // ...                            // two bases
public:
  void draw();                       // override Circle::draw()
};                                   //   and Triangle::draw()
```

8.4.3 Virtual Base Classes

This class hierarchy can be represented as:



Compare this diagram with the inheritance graph in §8.4.2 to see the difference between ordinary inheritance and virtual inheritance. In an inheritance graph, every base class that is specified to be *virtual* will be represented by a single object of that class. The language ensures that a constructor of a virtual base class is called exactly once.

8.4.4 Access Control in Multiple Inheritance

A name or a base class is called **accessible** if it can be reached through any path in a multiple inheritance graph, although often it can be reached through **several paths**.

Example:

```
class B {
public:
  double d;
  static double sdm;
};
class D1: public virtual B { /* ... */ };
class D2: public virtual B { /* ... */ };
class D12: protected D1, public D2 { /* ... */ };

D12* pd12 = new D12;
B* pb = pd12;              // accessible through public base D2
double m = pd12->d;        // accessible through public base D2
```

The member pt2d→d is accessible publicly since D2 has a public base D2 and d is a public member of D2. It would be inaccessible if both D1 and D2 were protected bases.

?? What if both D1 and D2 were public bases?

:: scope operator.

UNIVERSITY OF MANITOBA

8.4.4 Access Control in Multiple Inheritance

However, ambiguities might arise when a single entity was accessible through more than one path. Again, **the scope resolution operator** can be used to resolve such ambiguities. For instance,

Note that: There are two copies of member d in an object of X12, but there is only one static member *sdm* in X12.

```
class X1: public B { /* ... */ };
class X2: public B { /* ... */ };
class X12: protected X1, private X2 {
  void f();
};

void X12::f() {
  X12* p = new X12;
      // ... assign some value to *p
  double i = p->d;          // illegal, ambiguous
                            // X12::X1::B::d or X12::X2::B::d?

  double j = p->X1::B::d;   // OK
  double k = p->X2::B::d;   // OK

  double n = p->sdm;        // OK
                            // only one B::sdm in an X12 object
}
```

UNIVERSITY OF MANITOBA

Outline of Class Inheritance

- Introduction
- Derived Classes
- Abstract Classes
- Access Control
- Multiple Inheritance
- **Run-time Type Information**
- Replacing Virtual Functions by Static Polymorphism

Run-time Type Information

Problem:

Due to run-time polymorphism and assignment of pointers or references from one type to another, the type of an object may be lost. Recovering the run-time type information (or RTTI for short) of an object requires the system to examine the object to reveal its type.

Solution:

This section introduces two mechanisms that can be used to recover the run-time type of an object:

- 8.5.1 dynamic_cast Mechanism
- 8.5.2 typeid Mechanism

8.5.1 The dynamic_cast Mechanism

The **dynamic-cast** mechanism can be used to handle cases in which the correctness of a conversion can not be determined by the compiler.

The Mechanism:

Suppose p is a pointer,

dynamic_cast<T*>(p)

It will examines the object pointed to by p at run-time.

- Case1:(not nullptr) If this object is of class T or has a unique base class of Type T, then it returns a pointer of type T* to the object; Otherwise it return 0.
- Case2:(nullptr) If the value of p is 0, then dynamic_cast<T*>(p) returns 0.

8.5.1 The dynamic_cast Mechanism

Example:

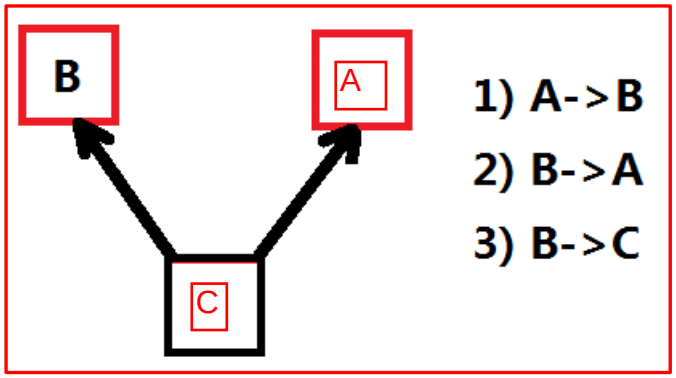
```
class A { /* ... */ };
class B { /* ... */ };
class C: public A, protected B { /* ... */ };
void f(C* p) {
  A* q1 = p;                          // OK
  A* q2 = dynamic_cast<C*>(p);        // OK

  B* p1 =p;                        // error, B is protected base
  B* p2 =dynamic_cast<C*>(p); // OK, but 0 is assigned to p2
}
void g(B* pb) {          // assume A, B have virtual functions
  if (A* pa = dynamic_cast<A*>(pb)) {
    // if pb points to an object of type A, do something
  } else {
    // if pb does not point to an A object, do something else
  }
}
```

1. Since B is protected base of C, function f() cannot directly convert a pointer to C into a pointer to B. Although using dymaic_cast makes it legal, the null pointer is returned.

In g(), converting B* into A* requires that A and B have virtual functions so that a variable of A* or B* may be of another type, for example, C*.

8.5.1 The dynamic_cast Mechanism

Upcast & Downcast:



Casting from a derived class to a base class is called an *upcast* because of the convention of drawing an inheritance graph growing from the base class downwards. Similarly, a cast from a base class to a derived class is called a *downcast*. A cast from a base to a sibling class (like the cast from B to A above) is called a *crosscast*. A *dynamic_cast* requires a pointer or reference to a polymorphic type to do a downcast or crosscast. The result of the return value of a *dynamic_cast* of a pointer should always be explicitly tested, as in function $g()$ above. If the result is 0, it means a failure in the conversion indicated.

UNIVERSITY OF MANITOBA

8.5.1 The dynamic_cast Mechanism

Exception Handling:

For a reference r, then

```
dynamic_cast<T&>(r);
```

tests to see if the object referred to by r is of type T. When it is not, it throws a *bad_cast* exception. To test a successful or failed cast using references, a suitable handler should be provided; see Chapter 9 for details.

```
void h(B* pb, B& rb) {  // assume A, B have virtual functions
  A* pa = dynamic_cast<A*>(pb);
  if (pa) {             // when $pa$ is not null pointer
    // if pb points to an A object, do something
  } else {              // when $pa$ is null pointer
    // if pb does not point to an A object, do something else
  }

  A& ra = dynamic_cast<A&>(rb);  // rb refers to an A object?
}
int main() {
  try{
    A* pa = new A;
    B* pb = new B;
    h(pa, *pa);
    h(pb, *pb);
  } catch (bad_cast) {          // exception handler
    // dynamic_cast<A&> failed, do something.
  }
}
```

8.5.1 The dynamic_cast Mechanism

Comparison:

A *dynamic_cast* can cast from a polymorphic virtual base class to a derived class or a sibling class, but a *static_cast* can not since it does not examine the object from which it casts. However, *dynamic_cast* can not cast from a *void** (but *static_cast* can) since nothing can be assumed about the memory pointed to by a *void**. Both *dynamic_cast* and *static_cast* can not cast away *const*, which requires a *const_cast*. For example,

```
void cast(const A* p, A* q) {
  q = const_cast<A*>(p);      // OK
  q = static_cast<A*>(p);     // error, can not cast away const
  q = dynamic_cast<A*>(p);    // error, can not cast away const
}
```

8.5.2 The typeid Mechanism

Problem:

The dynamic cast operator is enough for most problems that need to know the type of an object at run-time. However, situations occasionally arise when the exact type of an object needs to be known.

Solution:

The ***typeid*** operator yields an object representing the type of its operand; it returns a reference to a standard library class called ***type_info*** defined in header **<typeinfo>.** The operator ***typeid()*** can take a type name or an expression as its operand and returns a reference to a ***type_info*** object representing the type name or the type of the object denoted by the expression.

If the value of a pointer or a reference operand is 0, then ***typeid()*** throws a ***bad_typeid*** exception.

8.5.2 The typeid Mechanism

Example:

```
void f(Circle* p, Triangle& r, Circle_in_Triangle& ct) {
  typeid(r);          // type of object referred to by r
  typeid(*p);         // type of object pointed to by p

  if (typeid(ct) == typeid(Triangle) { /* ... */ }
  if (typeid(r) != typeid(*p) { /* ... */ }

  cout << typeid(r).name();      // print out type name
}
```

The function ***type_info::name()*** returns a character string representing the type name. Equality and inequality of objects of ***type_info*** can be compared.

There can be more than one ***type_info*** object for each type in a system. Thus comparisons for equality and inequality should be applied to ***type_info*** objects instead of their pointers or references.

UNIVERSITY OF MANITOBA

8.5.3 Run-Time Overhead

Recall:

The disadvantage of object-oriented programming is that it may sometimes greatly affect run-time performance.

Overhead:

Using run-time type information (*dynamic_cast* or *typeid*) involves overhead since the system examines the type of an object. It should and can be avoided in many cases in which virtual functions suffice. Use virtual functions rather than *dynamic_cast* or *typeid* if possible since virtual functions normally introduce less run-time overhead. Even a virtual function can cause a lot of performance penalty when it is small (i.e., it does not perform a lot of operations) and called frequently (e.g., inside a large loop). When a virtual function contains a lot of code (i.e., performs a lot of computation) or is not called frequently, the overhead of virtual function dispatch will be insignificant compared to the overall computation.

In large-scale computations, static (compile-time) checking may be preferred where applicable, since it is safer and imposes less overhead than run-time polymorphism. On one hand, run-time polymorphism is the cornerstone of object-oriented programming. On the other hand, too much run-time type checking can also be a disadvantage due to the run-time overhead it imposes, especially in performance-sensitive computations.

UNIVERSITY OF MANITOBA

Class Inheritance 4

Lecture 6

ECE7650-T13: C++ and Object-Oriented Numeric Computing for Engineers

Date: Marth 11th, 2016

Electrical and Computer Engineering

Outline of Class Inheritance

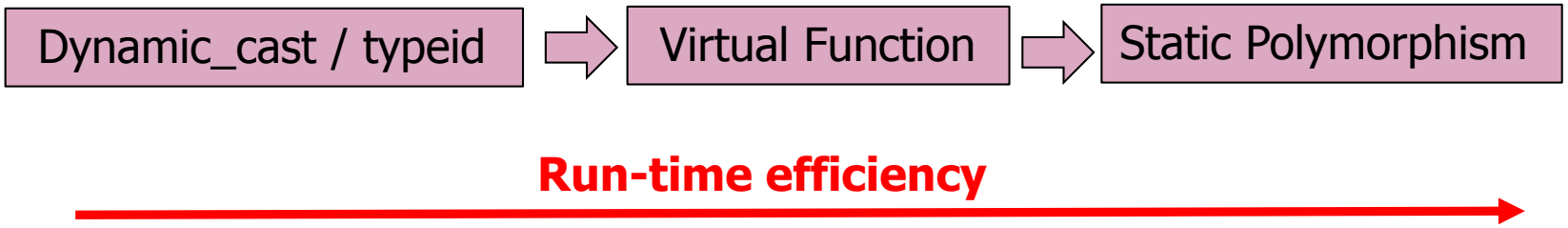
- Introduction
- Derived Classes
- Abstract Classes
- Access Control
- Multiple Inheritance
- Run-time Type Information
- **Replacing Virtual Functions by Static Polymorphism**

Static Polymorphism

- Introduction
- Method 1
- Method 2

Introduction

Virtual functions provide a nice design technique in software engineering. However, virtual functions may slow down a program when they are called frequently, especially when they do not contain a lot of instructions.



In this section, two techniques are presented to replace certain virtual functions by static polymorphism in order to improve **run-time efficiency**.

Method 1

Key Idea:

The idea of the first technique is to define a base template class and define functions operating on this base class. When defining a derived class from the base template class, the template parameter of this base class is taken to be the type of the derived class. Then the functions defined for the base class can be used for derived classes as well.

Example:

Questions: Consider the example in which a function CG() is defined for a base matrix class but may be called on derived classes for full, band and sparse matrices.

One Solution: A virtual function is used in §8.2 to defined matrix-vector multiplication upon which the function CG() depends. Then this virtual function is overridden in derived classes for full, band and sparse matrices, which enables one version of CG() to be called for different derived classes.

New Solution: Here the same goal can be achieved without using a virtual function. Rather, a matrix-vector multiplication is defined for a base template class that refers to the template parameter, which will be a derived class.

UNIVERSITY OF MANITOBA

Method 1

Code:

```
class Vtr {   /* a vector class */  };

template<class T> class Matrix {      // base class
  const T& ReferToDerived() const {
    return static_cast<const T&>(*this);
  }
public:

  // refer operation to derived class
  Vtr operator*(const Vtr& v) const {
    return ReferToDerived()*v;
  }

  // define common functionality in base class that can
  // be called on derived classes
  void CG(const Vtr& v) { // define CG here
    Vtr w = (*this)*v;    // call matrix vector multiply
    cout << "calling CG\n";
  }
};
```

UNIVERSITY OF MANITOBA

Method 1

Code:

```
class FullMatrix: public Matrix<FullMatrix> {
//   encapsulate storage information for full matrix
public:
  Vtr operator*(const Vtr& v) const {
    // define (full) matrix vector multiply here
    cout << "calling full matrix vector multiply\n";
  }
};

class BandMatrix: public Matrix<BandMatrix> {
//   encapsulate storage information for band matrix
public:
  Vtr operator*(const Vtr& v) const {
    // define (band) matrix vector multiply here
    cout << "calling band matrix vector multiply\n";
  }
};

class SparseMatrix: public Matrix<SparseMatrix> {
//   encapsulate storage information for sparse matrix
public:
  Vtr operator*(const Vtr& v) const {
    // define (sparse) matrix vector multiply here
    cout << "calling sparse matrix vector multiply\n";
   }
 };
```

Method 1

```
void f(const Vtr& v) {
  FullMatrix A;
  A.CG(v);            // Calling CG() on full matrix

  BandMatrix B;
  B.CG(v);            // Calling CG() on band matrix

  SparseMatrix S;
  S.CG(v);            // Calling CG() on sparse matrix
}
```

The function CG() is defined for the base class ***Matrix*** which depends on the matrix-vector multiply operator *. The operator * is syntactically defined for the base ***Matrix***, but actually refers to the operator * belonging to the template parameter **T**. When **T** is instantiated by, for example, ***FullMatrix*** in the definition of the class ***FullMatrix***, this operator * now is the **FullMatrix-vector** multiply operator which can be fully defined according to its matrix structure.

Comments:

In this approach, the type of an object is known at **compile-time** and thus there is no need for virtual function dispatch. One version of the function CG() can be called for different classes, which is achieved without using virtual functions.

Method 2

Key Idea:

The second technique in this section does not use inheritance at all, but only uses templates. It is much simpler in many situations. Taking the matrix solver CG() as an example , this technique define CG() as a global template function whose template parameter can be any type that defines a multiply operator with a vector.

UNIVERSITY OF MANITOBA

Method 2

```
class Vtr {   /* a vector class */  };
template<class M>   // define CG as a global function
void CG(const M& m, const Vtr& v) {
  // define CG here for matrix m and vector v
  Vtr w = m*v;       // call matrix vector multiply
  cout << "calling CG\n";
}
class FullMatrix {
//   encapsulate storage information for full matrix
public:
  Vtr operator*(const Vtr& v) const {
    // define (full) matrix vector multiply here
    cout << "calling full matrix vector multiply\n";
  }
};
class BandMatrix {
//   encapsulate storage information for band matrix
public:
  Vtr operator*(const Vtr& v) const {
    // define (band) matrix vector multiply here
    cout << "calling band matrix vector multiply\n";
  }
};
```

Method 2

```
class SparseMatrix {
//   encapsulate storage information for sparse matrix
public:
  Vtr operator*(const Vtr& v) const {
    // define (sparse) matrix vector multiply here
    cout << "calling sparse matrix vector multiply\n";
  }
};
void ff(const Vtr& v) {
  FullMatrix A;
  CG(A, v);           // Calling CG() on full matrix

  BandMatrix B;
  CG(B, v);           // Calling CG() on band matrix

  SparseMatrix S;
  CG(S, v);           // Calling CG() on sparse matrix
}
```

UNIVERSITY OF MANITOBA

Example2: Method 1 and Method 2

Example2 :

First define a function sum() for a base template matrix that adds all entries of a matrix.

Then define a derived class for full matrices that stores all entries of a matrix and another derived class for symmetric matrices that stores only the lower triangular part of a matrix to save memory.

The goal is to give only one version of sum() that can be called for full and symmetric matrices without using virtual functions.

Example2: Method 1

Method 1 :

```
template<class T> class Mtx {     // base matrix
private:
  // refer to derived class
  T& ReferToDerived() {
    return static_cast<T&>(*this);
  }

  // entry() uses features of derived class
  double& entry(int i, int j) {
    return ReferToDerived()(i,j);
  }
protected:
  int dimn;             // dimension of matrix
public:
  // define common functionality in base class that can
  // be called on derived classes
  double sum() {        // sum all entries
    double d = 0;
    for (int i = 0; i < dimn; i++)
      for (int j = 0; j < dimn; j++) d += entry(i,j);
    return d;
  }
};
```

UNIVERSITY OF MANITOBA

Example2: Method 1

Method 1 :

```
class FullMtx: public Mtx<FullMtx> {
  double** mx;
public:
  FullMtx(int n) {
    dimn = n;
    mx = new double* [dimn];
    for (int i=0; i<dimn; i++) mx[i] = new double [dimn];
    for (int i=0; i<dimn; i++)
      for (int j=0; j<dimn; j++)
        mx[i][j] = 0;                    // initialization
  }
  double& operator()(int i, int j) { return mx[i][j]; }
};
class SymmetricMtx: public Mtx<SymmetricMtx> {
  // store only lower triangular part to save memory
  double** mx;
public:
  SymmetricMtx(int n) {
    dimn = n;
    mx = new double* [dimn];
    for (int i=0; i<dimn; i++) mx[i] = new double [i+1];
    for (int i=0; i<dimn; i++)
      for (int j = 0; j <= i; j++)
        mx[i][j] = 0;                    // initialization
  }
  double& operator()(int i, int j) {
    if (i >= j ) return mx[i][j];
    else return mx[j][i];                // due to symmetry
  }
};
```

UNIVERSITY OF MANITOBA

Example2: Method 1

Method 1 :

```
void g() {
  FullMtx A(2);
  A(0,0) = 5; A(0,1) = 3; A(1,0) = 3; A(1,1) = 6;
  cout << "sum of full matrix A = " << A.sum() << '\n';

  SymmetricMtx S(2);   // just assign lower triangular part
  S(0,0) = 5; S(1,0) = 3; S(1,1) = 6;
  cout << "sum of symmetric matrix S = " << S.sum() << '\n';
}
```

Here the member function ***sum()*** depends on ***entry(i,j)*** that refers to an entry at row i and column j of a matrix through a function call operator (). But entry() is not defined for **FullMtx** and **SymmetricMtx**, and the function call operator () is not defined for the base class **Mtx**.

UNIVERSITY OF MANITOBA

Example2: Method 2

Method 2 :

```
template<class M> double sum(const M& m) {
  double d = 0;
  for (int i = 0; i < m.dimn; i++)
    for (int j = 0; j < m.dimn; j++) d += m.entry(i,j);
  return d;
}
class FullMtx {
  double** mx;
public:
  FullMtx(int n) {
    dimn = n;
    mx = new double* [dimn];
    for (int i=0; i<dimn; i++) mx[i] = new double [dimn];
    for (int i=0; i<dimn; i++)
      for (int j=0; j<dimn; j++)
        mx[i][j] = 0;                    // initialization
  }
  int dimn;
  double& operator()(int i, int j) { return mx[i][j]; }
  double entry(int i, int j) const {
    return const_cast<FullMtx&>(*this)(i,j);
  }
};
```

Example2: Method 2

Method 2 :

```
class SymmetricMtx {
  // store only lower triangular part to save memory
  double** mx;
public:
  SymmetricMtx(int n) {
    dimn = n;
    mx = new double* [dimn];
    for (int i=0; i<dimn; i++) mx[i] = new double [i+1];
    for (int i=0; i<dimn; i++)
      for (int j = 0; j <= i; j++)
        mx[i][j] = 0;                    // initialization
  }
  int dimn;
  double& operator()(int i, int j) {
    if (i >= j ) return mx[i][j];
    else return mx[j][i];                // due to symmetry
  }
  double entry(int i, int j) const {
    return const_cast<SymmetricMtx&>(*this)(i,j);
  }
};
```

UNIVERSITY OF MANITOBA

Example2: Method 2

Method 2 :

```
void gg() {
  FullMtx A(2);
  A(0,0) = 5; A(0,1) = 3; A(1,0) = 3; A(1,1) = 6;
  cout << "sum of full matrix A = " << sum(A) << '\n';

  SymmetricMtx S(2);   // just assign lower triangular part
  S(0,0) = 5; S(1,0) = 3; S(1,1) = 6;
  cout << "sum of symmetric matrix S = " << sum(S) << '\n';
}
```

When function entry() is large and accesses local information of **FullMtx** and **SymmetricMtx**, this may cause a lot of code duplication. In this case, the first technique might be better. This example is only for illustrating the idea if just for summing entries of a matrix the function entry() may not be needed.

Static Polymorphism

Conclusion:

Using virtual functions as shown in §8.2, the derived classes ***FullMatrix***, ***BandMatrix*** and ***SparseMatrix*** may not have to be known when the base class ***AbsMatrix*** is compiled. That is, if one wants to add one more class ***SymmetricMatrix*** to this class hierarchy, the code for the base class ***AbsMatrix*** need not be recompiled.

However, using the techniques here without virtual functions, the complete type of all classes must be known at compile-time. Thus, in this aspect, there may be some design disadvantage to static polymorphism. Besides, not all virtual functions can be replaced by templates.

Work fine→ efficient → extendibility, maintainability, readability (elegant)

UNIVERSITY OF MANITOBA

After the lecture

http://publib.boulder.ibm.com/iseries/v5r1/ic2924/books/c092712220.htm

The C++ programming language is based on the C language. Although C++ is a descendant of the C language, the two languages are not always compatible.

In C++, you can develop new data types that contain functional descriptions (member functions) as well as data representations. These new data types are called ***classes.*** The work of developing such classes is known as *data abstraction*. You can work with a combination of classes from established class libraries, develop your own classes, or derive new classes from existing classes by adding data descriptions and functions. New classes can contain (***inherit***) properties from one or more classes. The classes describe the data types and functions available, but they can hide (***encapsulate***) the implementation details from the client programs.

You can define a series of functions with different argument types that all use the same function name. This is called ***function overloading***. A function can have the same name and argument types in base and derived classes.

Declaring a class member function in a base class allows you to override its implementation in a derived class. If you use virtual functions, class-dependent behavior may be determined at run time. This ability to select functions at run time, depending on data types, is called ***polymorphism.***

You can redefine the meaning of the basic language operators so that they can perform operations on user-defined classes (new data types), in addition to operations on system-defined data types, such as int, char, and float. Adding properties to operators for new data types is called **operator overloading**.

The C++ language provides **templates** and several keywords not found in the C language. Other features include *try-catch-throw* **exception handling**, stricter type checking and more versatile access to data and functions compared to the C language.

UNIVERSITY

After the lecture

About C++

3 Main Festures	Techs & Library	Programming Style	OOP in high level
1. Encapsulation 2. Inheritance 3. Polymorphism	**Techs:** Templates, lambda/ regular expression, Exception handling **Libraries:** STL, Boost, xml, Socket program, thread, sequence	1. Procedural P.. 2. generic P.. 3. metaprogra.. 4. oop	1.Design Patterns (MVC,Singleton, Factory, viewer, Iterator...) 2. Frameworks.

Better to know about Dynamic programming languages: Python, Lua, Javascript, PHP, matlab, batch ....