LECTURE 10

Integration Using Numerical Recipes

Numerical Recipes in C: The Art of Scientific Computing by W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery (Cambridge Univ. Press, 1992) is an excellent text on scientfic computing techniques. They have a homepage at http://www.nr.com and at http://cfata2.harvard.edu/nr/ where you can download the text of the book. The book covers things like integration, root finding, interpolation, sorting, minimizing and maximizing functions, etc. They give a brief description of the technique and then the function (or subroutine if you use Fortran) that does the trick. I want to show how you can adapt their functions which are written in C to C++.

Let's take integration as an example. Suppose you want to integrate a function $f(x)$ over some interval from a to b.

$$I = \int_a^b f(x)dx \tag{1}$$

If you look up *Numerical Recipes in C*, then there will be functions which will calculate the integral of your function $f(x)$. How do you turn this into object–oriented code? In the example I give, I make an abstract base class `RealFunction` to be the generic function that is to be integrated. The actual function is derived from `RealFunction`. The intergration routines are also turned into classes with the definition of the function operator (`operator()()`) adapted from the integration function in *Numerical Recipes*. This is the general strategy. Let's see how it works in detail.

Trapezoidal Rule

The simplest way to do an integral is to use the trapezoidal rule and divide the abscissa into equally spaced intervals with a spacing h. Then you have a sequence of abscissas, denoted by x_0, x_1, x_2, ..., x_N. The values of the function at these x_i's is

$$f(x_i) \equiv f_i \tag{2}$$

The simple trapezoidal rule says that the integral for one of these intervals is given by

$$\int_{x_1}^{x_2} f(x)dx = h\left[\frac{1}{2}f_1 + \frac{1}{2}f_2\right] + O(h^3 f'') \tag{3}$$

If we use this equation $N-1$ times to do the integration in the intervals (x_1, x_2), (x_2, x_3), ..., (x_{N-1}, x_N), and then add the results, we obtain an "extended" formula for the integral from x_1 to x_N. This is the formula for the extended trapezoidal rule:

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{1}{2}f_1 + f_2 + f_3 + ...f_{N-1} + \frac{1}{2}f_N\right] + O\left(\frac{(b-a)^3 f''}{N^2}\right) \tag{4}$$

There are 2 functions given by Numerical Recipes (section 4.2) for the extended trapezoidal rule. `trapzd` is a routine that computes the nth stage of refinement of an extended trapezoidal rule. `func` is input as a pointer to the function to be integrated between limits a and b which are also input. When called with $n = 1$, the routine returns the crudest estimate of $\int_a^b f(x)dx$. Subsequent calls with $n = 2, 3, ...$(in that sequential order) will improve the accuracy by adding 2^{n-2} additional interior points.

```
#define FUNC(x) ((*func)(x))

float trapzd(float (*func)(float), float a, float b, int n)
{
    float x,tnm,sum,del;
    static float s;
    int it,j;

    if (n == 1) {
            return (s=0.5*(b-a)*(FUNC(a)+FUNC(b)));
    } else {
            for (it=1,j=1;j<n-1;j++) it <<= 1;
            tnm=it;
            del=(b-a)/tnm;              /*This is the spacing of points to be
                                          added. */
            x=a+0.5*del;
            for (sum=0.0,j=1;j<=it;j++,x+=del) sum += FUNC(x);
            s=0.5*(s+(b-a)*sum/tnm);  /*This replaces s by its refined value.*/
            return s;
    }
}
#undef FUNC
```

qtrap refines the trapezoidal rule so that you can specify the desired accuracy. The parameter EPS can be set to the desired fractional accuracy and JMAX can be set so that $2^{\texttt{JMAX}-1}$ is the maximum allowed number of steps.

```
#include <math.h>
#define EPS 1.0e-5
#define JMAX 20

float qtrap(float (*func)(float), float a, float b)
{
        float trapzd(float (*func)(float), float a, float b, int n);
        void nrerror(char error_text[]);
        int j;
        float s,olds;

        olds = -1.0e30;
        for (j=1;j<=JMAX;j++) {
                s=trapzd(func,a,b,j);
                if (j > 5)
                        if (fabs(s-olds) < EPS*fabs(olds) ||
```

```
                                (s == 0.0 && olds == 0.0)) return s;
                olds=s;
        }
        nrerror("Too many steps in routine qtrap");
        return 0.0;
}
#undef EPS
#undef JMAX
```

10^{-6} is about the smallest you should make EPS. If EPS is too small, `qtrap` takes too many steps to converge, and the accumulated roundoff error may prevent the routine from converging. The routine `qtrap` is not very sophisticated, but it's a fairly robust way of doing integrals of functions that are not very smooth. In other words, it's good for jagged functions.

Romberg Integration

If the function you are integrating over is fairly smooth, then the best way to do the integral is using Romberg's method. I think of this as approximating an interval of the function by a polynomial of order k where k is determined by the routine. `qromb` is a powerful and fast routine. It calls 3 other functions that are given by Numerical Recipes: `polint`, `trapzd`, and `nerror`.

```
#include <math.h>
#define EPS 1.0e-6
#define JMAX 20
#define JMAXP (JMAX+1)
#define K 5
#define NRANSI
/* #include "nrutil.h" */

float qromb(float (*func)(float), float a, float b)
{
void polint(float xa[], float ya[], int n, float x, float *y, float *dy);
float trapzd(float (*func)(float), float a, float b, int n);
void nrerror(char error_text[]);
float ss,dss;
float s[JMAXP],h[JMAXP+1];
int j;

h[1]=1.0;
for (j=1;j<=JMAX;j++) {
s[j]=trapzd(func,a,b,j);
if (j >= K) {
polint(&h[j-K],&s[j-K],K,0.0,&ss,&dss);
```

```
if (fabs(dss) <= EPS*fabs(ss)) return ss;
}
h[j+1]=0.25*h[j];
}
nrerror("Too many steps in routine qromb");
return 0.0;
}
/* ********************************************************************** */
void polint(float xa[], float ya[], int n, float x, float *y, float *dy)
{
int i,m,ns=1;
float den,dif,dift,ho,hp,w;
float *c,*d;

dif=fabs(x-xa[1]);
c=vector(1,n);
d=vector(1,n);
for (i=1;i<=n;i++) {
if ( (dift=fabs(x-xa[i])) < dif) {
ns=i;
dif=dift;
}
c[i]=ya[i];
d[i]=ya[i];
}
*y=ya[ns--];
for (m=1;m<n;m++) {
for (i=1;i<=n-m;i++) {
ho=xa[i]-x;
hp=xa[i+m]-x;
w=c[i+1]-d[i];
if ( (den=ho-hp) == 0.0)
                            nrerror("Error in routine polint");
den=w/den;
d[i]=hp*den;
c[i]=ho*den;
}
*y += (*dy=(2*ns < (n-m) ? c[ns+1] : d[ns--]));
}
        free_vector(d,1,n);
        free_vector(c,1,n);
}
```

```
/* ************************************************************** */
real *vector(long nl, long nh)
/* allocate a real vector with subscript range v[nl..nh]
can replace with "new" in C++ */
{
        real *v;

        v=(real *)malloc((size_t) ((nh-nl+1+NR_END)*sizeof(real)));
        if (!v) nrerror("allocation failure in vector()");
        return v-nl+NR_END;
}
/* ************************************************************** */
void free_vector(real *v, long nl, long nh)
/* free a real vector allocated with vector()
can replace with "delete" in C++ */
{
        free((FREE_ARG) (v+nl-NR_END));
}

#undef NRANSI
#undef EPS
#undef JMAX
#undef JMAXP
#undef K
```

C Program for Romberg and Trapezoidal Integration

You can take the Numerical Recipes subroutines as black boxes but you should always test them with a simple case to make sure you understand how they work and to make sure that they work right. There are occasional mistakes in the subroutines. Sometimes simple cases, like integrating a constant function, can reveal errors. So let's write a test C program to integrate the function from 0 to $\pi/2$.

$$f(x) = x^2(x^2 - 2.0)\sin(x) \tag{5}$$

We can integrate this analytically

$$\int_0^{\pi/2} f(x) = 4x(x^2 - 7)\sin(x) - (x^4 - 14x^2 + 28)\cos(x) \tag{6}$$

Having the exact answer allows us to test how accurate our numerical result is. The code is

```
/* Driver for routine qromb */
```

```
#include <stdio.h>
#include <math.h>
#define PIO2 1.5707963

/* Test function */
float func(float x)
{
        return x*x*(x*x-2.0)*sin(x);
}

/* Integral of test function func, i.e., test result */
float fint(float x)
{
        return 4.0*x*(x*x-7.0)*sin(x)-(pow(x,4.0)-14.0*x*x+28.0)*cos(x);
}

int main(void)
{
        float a=0.0,b=PIO2,s,t;

        printf("Integral of func computed with QROMB and QTRAP\n\n");
        printf("Actual value of integral is %12.6f\n",fint(b)-fint(a));
        s=qromb(func,a,b);
        printf("Result from routine QROMB is %11.6f\n",s);
        t=qtrap(func,a,b);
        printf("Result from routine QTRAP is %11.6f\n",t);
        return 0;
}
```

The result of running `main` is

```
Integral of func computed with QROMB and QTRAP

Actual value of integral is    -0.479158
Result from routine QROMB is   -0.479159
Result from routine QTRAP is   -0.479153
```

So it looks like `qromb` does a little better. One of the disadvantages of the C routines is that you can only integrate over one variable. If you want to integrate $f(x, y)$ over x and y, you have to make another copy of `qromb` (call it `qromb2`), and then use `qromb2` to integrate over `func2` which in turn calls `qromb`:

```
float func(float x, float y)
{
```

```
        return f(x,y);                  /* This is schematic */
}
float func2(float y)                    /* Fix y, integrate over x */
{
  return qromb(func,ax,bx);
}
int main(void)
{
  float integral=qromb2(func2,ay,by);
  return 0;
}
```

It's much more elegant if we use C++.

C++ Program for Romberg and Trapezoidal Integration

What we would like to do is make a class for the integration routine that can take any function. So we make each integration routine a class (`Romberg` and `Trapeze`) and the function to be integrated becomes a class derived from a pure abstract class `RealFunction`.

integration.h

```
#include <iostream.h>
#include <math.h>
typedef float real;

class RealFunction{                             //abstract class
public:
  virtual real operator()()=0;                 //pure virtual function
  virtual ~RealFunction(){};                   //virtual destructor
};

class Func1: public RealFunction{              //actual function to be integrated
public:                                        //This should be in testinteg.C
    real x;
    virtual real operator()()                  //overloads pure virtual function
     {return x*x*(x*x-2.0)*sin(x);}
};

class Trapeze: public RealFunction{
    real s;
    RealFunction &func;    //the generic function to be integrated
    real a,b;             //integral limits
    real &x;              //integration variable
    real EPS;             //desired accuracy
```

```
    int  JMAX;             //2**(JMAX)=max number of func. evaluation
public:
    virtual real operator()();             //overloads pure virtual function
    real operator()(real min,real max,     //redo integral with different
        real precision=1.0e-7,int max_iter=20){                //parameters
        a=min;b=max;EPS=precision;
        JMAX=max_iter; return (*this)();};
    Trapeze(RealFunction &f,real &Var,real min,real max,
                real precision=1.0e-7,int max_iter=20):func(f),x(Var)
        {a=min;b=max;EPS=precision;JMAX=max_iter;};
    real Trap(int n);        //workhorse that really does the integral
};

class Romberg : public RealFunction{
    real s;
    RealFunction &func;    //the function to integrate
    real a,b;            //integral limits
    real &x;             //integration variable
    real EPS;            //desired accuracy
public:
    virtual real operator()();
    real operator()(real min,real max,real precision=1.0e-7)
        {a=min;b=max;EPS=precision;return (*this)();};
    Romberg(RealFunction &f,real &Var,real min,real max,
                real precision=1.0e-7):func(f),x(Var)
        {a=min;b=max;EPS=precision;};
    real midpnt(int n);    //workhorse that does the integral (open bc)
};

void polint(real xa[],real ya[],int n,real x,real &y,real &dy);
```

Integration.C

These member functions have been adapted from the subroutines in Numerical Recipes in C. I have marked the lines which have been changed.

```
#include "integration.h"
#include <iostream.h>
#define JMAXP 14
#define K 5

real Trapeze::operator()() {                 //adapted from qtrap

    real s,olds;
```

```
    int j;

    olds = -1.0e30;
    for (j=1;j<=JMAX;j++){
s=Trap(j);                          //changed; Trap is integrating fcn
        if (fabs(s-olds) <= EPS*fabs(olds)) return s;
if(s==0.0 && olds == 0.0 && j>6 ) return s;
        olds=s;
    }
    printf("Too many steps in routine qtrap_y\n");
    exit(1);
    return 0.0;
}
real Trapeze::Trap(int n) {               //adapted from trapzd

    real tnm,sum,del;
    int j,it;

    if (n == 1){
it=1;
x=a; s=func();                    //changed
x=b; s+=func();                   //changed
return (s*=0.5*(b-a));
    }else{
        for (it=1,j=1;j<n-1;j++) it <<=1;
        tnm=it;
        del=(b-a)/tnm;
        x=a+0.5*del;
        for (sum=0.0,j=1;j<=it;j++,x+=del) sum+=func();   //changed
        s=0.5*(s+(b-a)*sum/tnm);
        return s;
    }
}
real Romberg::operator()(){                //adapted from qromb
    int j;
    real ss,dss,h[JMAXP+2],s[JMAXP+2];
    h[1]=1.0;
    for(j=1;j<=JMAXP;j++){
s[j]=midpnt(j);                     //changed; midpnt is integrating
if(j>=K){                                                //function
    polint(&h[j-K],&s[j-K],K,0.0,ss,dss);
    if(fabs(dss)<=EPS*fabs(ss)) return ss;
```

```
}
s[j+1]=s[j];
h[j+1]=h[j]/9.0;
    }
    cout << "Too many steps in routine Romberg" << endl; exit(1);
    return 0.0;
}
real Romberg::midpnt(int n){                        //adapted from midpoint
    real tnm,sum,del,ddel;
    int it,j;

    if(n==1){
x=0.5*(a+b);                                 //changed; set x
return (s=(b-a)*func());                     //changed; evaluate func
    }else{
for(it=1,j=1;j<n-1;j++) it *=3;
tnm=it;
del=(b-a)/(3.0*tnm);
ddel=del+del;
x=a+0.5*del;
sum=0.0;
for(j=1;j<=it;j++){
    sum+=func();                            //changed; evaluate func
    x+=ddel;                                //changed; set x
    sum+=func();                            //changed; evaluate func
    x+=del;                                 //changed; set x
}
s=(s+(b-a)*sum/tnm)/3.0;
return s;
    }
}
```

polint.C

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
typedef float real;

real *vector(int nl,int nh);
void polint(real xa[],real ya[],int n,real x,real &y,real &dy)
{
   int i,m,ns=1;
```

```
   real den,dif,dift,ho,hp,w;
   real *c,*d;

   dif=fabs(x-xa[1]);
   c=vector(1,n);
   d=vector(1,n);
   for(i=1;i<=n;i++){
if((dift=fabs(x-xa[i])) < dif){
    ns=i;
    dif=dift;
}
c[i]=ya[i];
d[i]=ya[i];
    }
    y=ya[ns--];
    for(m=1;m<n;m++){
for(i=1;i<=n-m;i++){
   ho=xa[i]-x;
   hp=xa[i+m]-x;
   w=c[i+1]-d[i];
   if((den=ho-hp)==0.0) {printf("error in routine polint\n");
 exit(1);}
   den=w/den;
   d[i]=hp*den;
   c[i]=ho*den;
}
y += (dy=(2*ns < (n-m) ? c[ns+1] : d[ns--]));
   }
   delete [] d;                    //replaces free_vector
   delete [] c;                    //replaces free_vector
}

#define NR_END 1
real *vector(int nl,int nh)
// allocate a real vector with subscript range v[nl .. nh]
   {
    real *v;
    v=new real[nh-nl+1+NR_END];
    if(!v) error("allocation failure in vector1()");
    return v-nl+NR_END;
   }
```

```
void error(const char *s)
    {
    cerr << endl << s << endl; //1st "endl" is in case program is printing
    cout << endl << s << endl; //something out when the error occurs
    cout.flush();              //write the output buffer to the screen
                               //or wherever the output of cout goes.
    abort();
    }
#undef NR_END
```

Here is the test program for `main`:

```
#include <iostream.h>
#include "integration.h"
typedef float real;

int main() {
float integral, a=0.0, b=1.5707963;
Func1 Testfunc;                 //call default constructor
Trapeze Trap(Testfunc,Testfunc.x,a,b);
float integralt= Trap();
cout << "C++ Trapezoidal Integration Result = " << integralt << endl;
Romberg Rom(Testfunc,Testfunc.x,a,b);
integral=Rom();
cout << "C++ Romberg Integration Result = " << integral << endl;
return 0;
```

To compile we type

```
g++ -lm integration.C polint.C testinteg.C
```

The result is:

```
C++ Trapezoidal Integration Result = -0.479153
C++ Romberg Integration Result = -0.479159
```

We have defined an abstract class `RealFunction` which has a pure virtual function

```
virtual real operator()()=0;              //pure virtual function
```

This acts as a placeholder for the function that will be integrated. We tell the integration routine that it will be integrating a `RealFunction`. Then we derive the actual function from `RealFunction`. The function and the variable of integration are passed by reference as an argument of the constructor. In the integration subroutine, the `FUNC(a)` which appeared in the C program is replaced by

```
  x=a; s=func();
```

in the C++ program. The arguments that were passed to the integration function in the C program have become data members of the class. Most of the variables used in the integration subroutine do not need to be made class members, but the input and output parameters should be made class members. In addition, if there are static variables in the C program, these should be made (nonstatic) members of the class. In the C subroutines from Numerical Recipes, the static variables are ones that retain their values between different calls to the function. (Normally, the variables local to a function lose their values when the function goes out of scope.) By making them nonstatic data members of the class, they will keep their values between different calls to the member functions. In fact they will keep their values for as long as the object exists. (These values are not `const` and can be changed by the member functions.)

We have rewritten the C function `vector` so that it uses `new`. We have replaced the `free_vector` function with `delete`.

Revised C++ Code for Integration

`Romberg::midpnt(int n)` and `Trapeze::trap(int n)` are examples of what I will call "integrators". They are subroutines that really do the integration. Let's make `Romberg` and `Trapeze` more flexible so that they can easily accept any integrator. We can do this by making them template classes which accept the type of integrator. Integrator will be a base class with `midpnt` and `trap` derived from it.

We will also replace the error statement with throwing and catching exceptions.

integration.h

```
#include <math.h>
#include <iostream.h>
#include "exception.h"
#define JMAXP 14
#define K 5

typedef float real;

class RealFunction{                              //abstract class
public:
  virtual real operator()()=0;                  //pure virtual function
  virtual ~RealFunction(){};                    //virtual destructor
};

class Func1: public RealFunction{                //test function
public:
    real x;
    virtual real operator()()
     {return x*x*(x*x-2.0)*sin(x);}
```

```
};

class FuncXY: public RealFunction{           //another test function
public:
    real x, y;
    virtual real operator()()
     {return log(x*y);}
};

class Integrator{               //base class for the workhorse
                                //of the integration routine
public:
    real s;                     //value of integral
    RealFunction &func;         //the function to integrate
    real a, b;                  //integral limits
    real &x;                    //integration variable
    Integrator(RealFunction &f,real &Var,real min,real max):func(f),x(Var)
      {a=min;b=max;}
    SetLimits(real min, real max) {a=min;b=max;}
};

class Trapzd: public Integrator{
public:
   Trapzd(RealFunction &f,real &Var,real min,real max):
      Integrator(f,Var,min,max){};      //call base class constructor
   real operator()(int n);              //integration subroutine
};

class Midpoint: public Integrator{
public:
   Midpoint(RealFunction &f,real &Var,real min,real max):
      Integrator(f,Var,min,max){};      //call base class constructor
   real operator()(int n);              //integration subroutine
};

template<class SomeIntegrator>
class Trapeze: public RealFunction{
    real EPS;              //Desired precision
    int  JMAX;             //2**(JMAX)=max number of func. evaluation
    SomeIntegrator integrator;     //contains integration routine
public:
    virtual real operator()();
```

```
    real operator()        //Reset integration limits and accuracy
        (real min,real max,real precision=1.0e-7,int max_iter=20){
        integrator.SetLimits(min,max);
        EPS=precision;
        JMAX=max_iter;
        return (*this)();}
    Trapeze(RealFunction &f,real &Var,real min,real max,
        real precision=1.0e-7,int max_iter=20):
        integrator(f, Var, min, max)
        {EPS=precision;JMAX=max_iter;}
};

template<class SomeIntegrator>
class Romberg : public RealFunction{
    real EPS;              //Desired precision
    SomeIntegrator integrator;     //Integration routine
public:
    virtual real operator()();
    real operator()        //Reset integration limits and accuracy
        (real min,real max,real precision=1.0e-7)
        {integrator.SetLimits(min,max);
         EPS=precision;
         return (*this)();}
    Romberg(RealFunction &f,real &Var,real min,real max,
        real precision=1.0e-7):
        integrator(f, Var, min, max)
        {EPS=precision;}
};

template<class SomeIntegrator>
real Trapeze<SomeIntegrator>::operator()() {
    real s,olds;
    int j;

    olds = -1.0e30;
    for (j=1;j<=JMAX;j++){
        s=integrator(j);                      //call integration subroutine
        if (fabs(s-olds) <= EPS*fabs(olds)) return s;
if(s==0.0 && olds == 0.0 && j>6 ) return s;
        olds=s;
    }
    throw Exception("Too many steps in Trapeze()",__FILE__,__LINE__);
```

```
    return 0.0;
}

template<class SomeIntegrator>
real Romberg<SomeIntegrator>::operator()(){
    int j;
    real ss,dss,h[JMAXP+2],s[JMAXP+2];

    h[1]=1.0;
    for(j=1;j<=JMAXP;j++){
        s[j]=integrator(j);                    //call integration subroutine
        if(j>=K){
            polint(&h[j-K],&s[j-K],K,0.0,ss,dss);
            if(fabs(dss)<=EPS*fabs(ss)) return ss;
}
s[j+1]=s[j];
h[j+1]=h[j]/9.0;
    }
    throw Exception("Too many steps in Romberg",__FILE__,__LINE__);
    return 0.0;
}

void polint(real xa[],real ya[],int n,real x,real &y,real &dy);
```

exception.h

```
#include<iostream.h>
class Exception
{
  public:
    Exception(char* pMsg, char* pFile, int nLine)
    {
       strcpy(msg,pMsg);
       msg[sizeof(msg)-1]='\0';   //make sure msg is terminated
       strcpy(file,pFile);
       file[sizeof(file)-1]='\0';
       lineNum=nLine;
    }
    virtual void display(ostream& out)
    {
       out << "Error: " << msg << endl;
       out << "Occurred on line number " << lineNum << ", file "
           << endl;
```

```
    }
  protected:
    // error message
    char msg[80];
    // file name and line number where error occurred
    char file[80];
    int lineNum;
};
```

integration.C

```
#include <iostream.h>
#include "integration.h"

real Trapzd::operator()(int n) {
    real tnm,sum,del;
    int j,it;

    if (n == 1){
it=1;
x=a; s=func();                  //set x; evaluate func()
x=b; s+=func();                 //set x; evaluate func()
return (s*=0.5*(b-a));
    }else{
        for (it=1,j=1;j<n-1;j++) it <<=1;
tnm=it;
del=(b-a)/tnm;
x=a+0.5*del;
for (sum=0.0,j=1;j<=it;j++,x += del)  //set x
            sum+=func();                      //evaluate func()
s=0.5*(s+(b-a)*sum/tnm);
return s;
    }
}
// ********************************************************************
real Midpoint::operator()(int n){
    real tnm,sum,del,ddel;
    int it,j;

    if(n==1){
x=0.5*(a+b);                            //set x
return (s=(b-a)*func());                //evaluate func()
    }else{
```

```
for(it=1,j=1;j<n-1;j++) it *=3;
tnm=it;
del=(b-a)/(3.0*tnm);
ddel=del+del;
x=a+0.5*del;
sum=0.0;
for(j=1;j<=it;j++){
    sum+=func();                    //evaluate func()
    x+=ddel;                        //set x
    sum+=func();                    //evaluate func()
    x+=del;                         //set x
}
s=(s+(b-a)*sum/tnm)/3.0;
return s;
    }
}
```

polint.C

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include "exception.h"
typedef float real;

real *vector(int nl,int nh);
void polint(real xa[],real ya[],int n,real x,real &y,real &dy)
{
   int i,m,ns=1;
   real den,dif,dift,ho,hp,w;
   real *c,*d;

   dif=fabs(x-xa[1]);
   c=vector(1,n);
   d=vector(1,n);
   for(i=1;i<=n;i++){
if((dift=fabs(x-xa[i])) < dif){
    ns=i;
    dif=dift;
}
c[i]=ya[i];
d[i]=ya[i];
    }
```

```
    y=ya[ns--];
    for(m=1;m<n;m++){
for(i=1;i<=n-m;i++){
   ho=xa[i]-x;
   hp=xa[i+m]-x;
   w=c[i+1]-d[i];
   if((den=ho-hp)==0.0)
             {throw Exception("error in routine polint",__FILE__,__LINE__);}
   den=w/den;
   d[i]=hp*den;
   c[i]=ho*den;
}
y += (dy=(2*ns < (n-m) ? c[ns+1] : d[ns--]));
   }
   delete [] d;
   delete [] c;
}
#define NR_END 1
real *vector(int nl,int nh)
// allocate a real vector with subscript range v[nl .. nh]
   {
    real *v;
    v=new real[nh-nl+1+NR_END];
    if(!v)
    {throw Exception("allocation failure in vector()", __FILE__,__LINE__);}
    return v-nl+NR_END;
   }
#undef NR_END
```

We have made template classes out of Romberg and Trapeze to allow for using different integrators. We have made `Integrator` a base class, and the actual integrators are derived from `Integrator`. We chose not to make `Integrator` an abstract class with pure virtual functions since this costs overhead and doesn't really buy us anything. Notice that in `Romberg` and `Trapeze`, the line

```
s=integrator(j)
```

calls the integration subroutine. `integrator(j)` does not overload a virtual function in the base class `Integrator`. There is no function `Integrator::operator()(int j)`. Rather `integrator` is an instantiation of the template class `SomeIntegrator`. When we replace the template type `SomeIntegrator` with a concrete type, like `Romberg` or `Trapeze`, that concrete type had better have a member function `operator()(int j)`.

The function being integrated (`func`), the variable of integration, and the limits of integration are only needed by the integrators, and not by `Romberg` and `Trapeze` per se.

So we have made `func`, `x`, and the limits of integration data members of `Integrator`; they are no longer data members of `Romberg` and `Trapeze`. But the constructors of `Romberg` and `Trapeze` take these as arguments and pass them to `Integrator`.

testinteg.C

Here is the test program for our integration routines:

```
#include <iostream.h>
#include "integration.h"

int main() {
  try
     {
       float a=0.0, b=1.5707963;
       Func1 Testfunc;                  //call default constructor
       Trapeze<Trapzd> Trap(Testfunc,Testfunc.x,a,b);
       float integralt= Trap();
       Romberg<Midpoint> RomTest(Testfunc,Testfunc.x,a,b);
       float integral=RomTest();
       cout << "C++ Trapezoidal Integration Result = " << integralt << endl;
       cout << "C++ Romberg Integration Result = " << integral << endl;

// Now let's do a double integral over x and y.
       FuncXY FXY;
       a=0.05;
       b=9.0;
       Romberg<Midpoint> RomX(FXY,FXY.x,a,b);
       Romberg<Midpoint> Rom(RomX,FXY.y,a,b);
       integral=Rom();
       cout << "C++ Romberg XY Integration Result with Midpoint integrator= "
       << integral << endl;

       Romberg<Trapzd> TRomX(FXY,FXY.x,a,b);
       Romberg<Trapzd> TRom(TRomX,FXY.y,a,b);
       integral=TRom();
       cout << "C++ Romberg XY Integration Result with Trapzd integrator= "
       << integral << endl;
    }
     catch(Exception& x)
      {
        //use the built-in display member function
          x.display(cerr);
      }
```

```
        return 0;
}
```

The result of running this program is

```
C++ Trapezoidal Integration Result = -0.479153
C++ Romberg Integration Result = -0.479159
C++ Romberg XY Integration Result with Midpoint integrator= 196.451
C++ Romberg XY Integration Result with Trapzd integrator= 196.406
```

Notice that in the test program, we can easily do a double integral since `Romberg` and `Trapeze` overload the pure virtual function `virtual real operator()()=0` in class `RealFunction`. In other words, `Romberg` and `Trapeze` are classes derived from `RealFunction`. So `Romberg IS_A RealFunction` and `Trapeze IS_A RealFunction`. Their output is completely acceptable as a function that can be integrated yet again.