CHAPTER 7
MACROS FOR GENERIC CLASSES

7.1 Introduction

Prior to the introduction of the template paradigm into C++, generic classes were created via a quite intricate and sophisticated use of the preprocessor. This was done using what is called **#define** macros**.** These macros are not actual C or C++ statements; rather they are expanded into C or C++ language statements by the preprocessor. Some simple macros are:

```
#define  CALLOC(N,X)    (X*) calloc(N ,  sizeof(X))

#define  MAX(A,B)   ( ( A>B )  ?   A  :  B  )
```

They could be used in a function like this :

```
main()  {
    int*  iptr  =  CALLOC(100,  int);   // this allocates contiguous   memory   for  100
    //   ints
    //    and  takes  the  starting  address  of  this  memory   and  assigns  it  to  iptr.
    int   m = 5,  n  = 7,  p;
    p  =  MAX(m , n);
};
```

After the two statements inside **main** are expanded at compile time by the preprocessor, **main** reads:

```
main()  {
    int*  iptr =  (int*)calloc(100,   sizeof(int));
    int   m = 5,  n = 7,  p;
    p  =  ( (m>n)   ?   m  :  n );
};
```

After the preprocessor finishes, the preprocessed code will be compiled.

Notice how both the **MAX** macro and the **CALLOC** macro are not strongly typed. We can use **CALLOC** with any type **X** and we can use **MAX** with any variables **A**, **B** that can be compared with the **operator<**. It should be noted that in C++, we use the operator **new** in place of **calloc** and **CALLOC**.

Another property of macros is that the preprocessor replaces the macro call with actual expanded code, not a function call. It is true that we could write a maximum function for each of the relevant classes, and then call the proper **max** function when necessary. What would that entail? Essentially we would duplicate and modify the source code for **max** many times. This would allow us to call a function **max** and get the correct version of **((m > n) ? m : n)** executed. In effect, the macro serves to inline this statement and eliminate the time overhead of a function call. Thus, macros in C and C++ do have advantages and disadvantages. We like inlining; this is the reason why C++ has the inlining facility built into it. Macros are no longer used in C++ to do pure inlining; we use the built-in inline facility. In the class definition, if a function prototype appears with its code body, it is inlined if possible. If we do not put the body of a function with its prototype in the class definition, then we can still inline it by putting the word **inline** in front of the normal prototype in the class definition. The disadvantages of macros are that (1) they circumvent strong typing and (2) they can cause unexpected semantic problems when expanded.

7.2 A Macro for Generic Vectors

The memory management of a vector class is completely independent of the underlying number system. The structure is the same whether it is for **doubles** or **ints**. This section will demonstrate the intricate use of macros to produce the source code for each vector class we desire. By making the appropriate directives we can get the preprocessor to generate the C++ source for both the class definition and the implementation files. Getting all the code to do this successfully is a painful process. Luckily, for the most part we can rely on the C++ **template** paradigm to replace macros in almost all cases. Since **templates** may fall short in some special situation, the generic macro approach will be discussed here.

In order to understand the approach, the full files have to be displayed. The file containing the **main** function is **testvec.cpp**:

```
#include "myvector.h"
declare(vectorrep, int);
declare(vector, int);
int main() {
    vectorint  vi(3), wi(3), zi(3);
    vi[0] = 1;
    vi[1] = 10;
    vi[2] = 100;
    wi[0] = 10;
    wi[1] = 20;
    wi[2] = 30;
    zi = vi + wi;
    cout << "\n result is " << zi <<  "\n";
    cout << vi; cout << wi;
    cout << "\nenter the size of desired input ";
    int dim; cin >> dim;
return 1;
}
```

The first thing to mention is that we will call our classes of interest **vectorint**, **vectorfloat**, etc. We must make the decision what class names we desire at the beginning of the design of the macro. An extra little problem crept into this design. The preprocessor as far as the author is concerned performs in undocumented ways and obviously this will vary from implementation to implementation. The vector classes that are used throughout the book have the **vecrep struct** encapsulated inside of the vector class definition. The author had problems with the Borland implementation on this issue; there was no problem on the ATT 2.1 compiler with this. With macros, you cannot be sure if lack of success is the fault of the programmer or of the implementation. We determine that we will call the various **reps** names such as **vectorrepint**, **vectorrep-float**, etc.

Notice near the top of the above file we have the two statements:

```
declare(vectorrep,int);
declare(vector,int);
```

These two statements are macro calls of the **declare** macro which is found in the system header file: **generic.h**. **generic.h** is included in **myvector.h** which is included in **testvec.cpp**. Now, when the **main** file is preprocessed the first macro will be expanded, and if its expansion contains any macros, they will be expanded. This continues until the original **declare(vectorrep, int)** yields pure C++ source. Then the preprocessor will start the expansion process on the second **declare**. Let's take a look at the **declare** macro:

```
#define  declare(a,t)  name2(a, declare) (t)
```

This means **declare(vectorrep, int)** will be expanded to:

```
name2(vectorrep, declare)(int)
```

What is **name2**? Its job is to concatenate its two arguments. One should consult Ellis and Stroustrup[] for a good discussion of the preprocessor. Also, Borland uses a macro called **_Paste2()** instead of **name2()**. Now, **name2(vectorrep, declare)** will be expanded to **vectorrep-declare**. Thus, the expansion finally yielded from the **generic.h** macros is:

```
vectorrepdeclare(int)
```

7.3 Header File for the Vector Classes

Our job is to write a macro that expands this and puts it in the header file that we will use when working with generic vectors. Again this whole process is difficult. The entire header file **myvector.h** will be listed now:

```
#ifndef MYVECTOR_H
#define MYVECTOR_H
#include <stdlib.h>
#include <iostream.h>
#include <stdio.h>
#include <generic.h>
#include <stddef.h>
void err_msg(char *p);
#define vecrep(T) _Paste2(vecrep,T)
#define vector(T) _Paste2(vector,T)
#define vectorrepdeclare(T) \
struct vecrep(T) \
{ \
     T *v; \
     int sz; \
     int refs; \
};

#define vectordeclare(T)\
class vector(T) \
{ \
private: \
     vecrep(T) *p; \
protected: \
     struct vecrep(T)* address_vecrep() const {return p;} \
     T* address_array() const  { return p->v;} \
public: \
     vector(T)(int); \
     vector(T)(const vector(T)&); \
     vector(T)(); \
     vector(T)(); \
     int size() const  {  return p->sz; } \
     int refs() const  {  return p->refs; } \
     void assign(int n, T* mvptr); \
     T& elem(int i) const {return p->v[i];} \
     T& operator[]  (int) const; \
     vector(T)& operator= (const vector(T) &a); \
     void deepcopy(const vector(T)&); \
     vector(T) operator+(const vector(T)& a);\
     friend ostream& operator<<(ostream&,vector(T));\
};
#endif MYVECTOR_H
```

Several lines into the file we see:

```
#define  vectorrepdeclare(T)\
struct  vecrep<T>  \
{\
     T*  v;\
     int  sz;\
     int  refs;\
};
```

Notice how a backslash appears after all but the last line; this is used to indicate that the macro is being continued on the next line. A line with no terminating backslash indicates the end of the macro. This first macro will be expanded into:

```
struct vecrep(int) {
     int  *v;
     int  sz;
```

```
    int   refs;
};
```

Now, **vecrep(int)** is not an acceptable type name. This is taken care of by the macro:

```
    #define vecrep(int)  _Paste2(vecrep,T) put in above the declare macros.
```

This macro will expand **vecrep(int)** into **vecrepint**. It turns out this type is internal to the implementation, and thus we don't have to use it in **main**.

For the statement, **declare(vector, int)** in **main**, the macros in **generic.h** expand this into **vectordeclare(int)**, and thus we need another macro in our header file to expand this. When this **vectordeclare(T)** macro is expanded for this we get:

```
class  vector(int)  {
private:
    vecrep(int)* p;
protected:
    struct   vecrep(int)*  address_vecrep()   const  { return  p; }
    int*  address_array()  const  {return  p->v; }
public:
    vector(int)(int);
    vector(int)(const  vector(int)&);
    ...
};
```

Then the **vecrep(T)** and **vector(T)** macros will be expanded to yield:

```
class  vectorint  {
private:
    vecrepint* p;
protected:
    struct   vecrepint*  address_vecrep()   const  { return  p; }
    int*  address_array()  const  {return  p->v; }
public:
    vectorint(int);
    vectorint(const  vectorint&);
    ...
};
```

The way we use this header file of generic macros is to include them in any file where we want to use objects of a class that can be created with them, and then below all other preproces-sor type directives (**#defines**, **#includes**) we should put directives such as:

```
    declare(vectorrep, int);
    declare(vector, int);
    declare(vectorrep, float);
    declare(vector, float);
```

This will create the proper class definitions for vectors of floats and ints in this case. Then in these files we use the **class** name that we know will be produced. For the above, they are: (1) **vectorrepint**, (2) **vectorint**, (3) **vectorrepfloat**, and (4) **vectorfloat**.

7.4 Implementation File for the Vectors

The implementations not found in a class definition must be put into an implementation file. Therefore, we need to create a style of implementation file that consists of macros. For this particular class, the corresponding implementation file is:

```
#include <iostream.h>
void err_msg(char* p)
{
    cerr << p  << "\n";
}
#include "myvector.h"
#define vectorimplement(T) \
vector(T)& vector(T)::operator= (const vector(T) &a) \
{ \
    if (--p->refs == 0) \
    { \
        delete[] p->v; \
        delete p; \
    } \
    a.p->refs++; \
    p = a.p; \
    return(*this); \
} \
vector(T)::vector(T) (int s) \
{ \
    if (s<=0) err_msg("bad size vector in constructor"); \
    p = new vecrep(T); \
    p->sz = s; \
    p->v = new T[s]; \
    p->refs = 1; \
} \
vector(T)::vector(T) () \
{ \
    p = new vecrep(T); \
    p->sz = 0; \
    p->v = NULL; \
    p->refs = 1; \
} \
vector(T)::vector(T) (const vector(T) &mv) \
{ \
    mv.p->refs++; \
    p = mv.p; \
} \
void vector(T)::assign(int n, T* mnptr)  \
{ \
    if ( (p->sz !=0)||(p->v)) err_msg("you are assigning to nonvoid vector"); \
    else \
    { \
        p->v = mnptr; \
        p->sz = n; \
    } \
} \
T& vector(T)::operator[](int i) const \
{ \
    if (i < 0 || i >= p->sz) err_msg("vector index out of range"); \
    return p->v[i]; \
} \
vector(T)::vector(T)() \
{ \
    if (--p->refs == 0) \
    { \
        delete[] p->v; \
        delete p; \
    } \
} \
vector(T) vector(T)::operator+(const vector(T)  &a) \
{ \
    int r = a.size(); \
    int s = size(); \
    if ( r != s ) err_msg("vectors of diff dim in + "); \
    vector(T) sum(r); \
    for (int i = 0; i < r; i++) sum[i] = (*this)[i] + a[i]; \
    return sum; \
} \
ostream& operator<< (ostream& s,vector(T) mv) \
{ \
    int n = mv.size(); \
    s << "\n"; \
    for (int i = 0; i < n; i++) s <<  mv.elem(i) << "   "; \
    return(s); \
};
declare(vectorrep, int);
declare(vector, int);
implement(vector, int);
```

The immediately above **declares** generate the class definitions, and then we have **implement(vector, int)** which will be expanded by the appropriate macros in **generic.h** to yield the statement **vectorimplement(int)**. Then the above macro **vectorimplement(T)** will yield member function definitions with **vector(int)** and **vecrep(int) class** names. These will then be replaced with legal **vectorint** and **vecrepint** identifiers, and thus we will have valid member function implementations. This compiled file will provide the object code for the rest of the member functions at link time.

In order to be complete, let's go over the **implement** macro. It has the same structure as **declare**:

```
#define   implement(a, t)   name2(a,  implement)(t)
```

Thus **implement(vector, int)** yields **vectorimplement(int)**.

See the listings at the end of this chapter for a valid **main** file.

7.5 Encapsulation of vecrep

In the above, the Borland compiler did not like a nested class definition for the **vecrep** class inside the **vector** class, and thus it was pulled out. The ATT 2.1 compiler accepted the nested macroized definition with no problem. When working with macroized classes, many problems seem to arise before success. If possible, the **template** paradigm should be used in place of macros.

For the **template** paradigm, it should be noted that on an SGI 3.0 compiler there was a similar problem with an encapsulated **vecrep<T>** in **vector<T>**, but no problem doing this in Borland. This is the opposite of macros. The use of templates is much more stable, and it is reasonable to assume that the author was not at fault with this.

7.5 C++ Files

7.5.1 Generic Modular Arithmetic with Vector Templates

7.5.1.1 File gen_mod2.h

```
//  the interface to class for generic modulo numbers

#ifndef GENERIC_MOD_CLASS_H
#define GENERIC_MOD_CLASS_H
#include <generic.h>
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>

typedef long INTEGER;

#define mnum(ENYENT) name2(mnum,ENYENT)

#define modulodeclare(ENYENT) \
class mnum(ENYENT) \
{ \
   private: \
     INTEGER coset; \
     static INTEGER divisor; \
     void simplify(void) \
     { \
        INTEGER temp = coset % divisor; \
        if (temp < 0) temp += divisor; \
        coset = temp; \
     } \
     static INTEGER gcd(INTEGER,INTEGER); \
     inline static INTEGER LABS(INTEGER);\
   public: \
     mnum(ENYENT)(int dividend)\
     {\
          coset = dividend;\
          simplify();\
     }\
     mnum(ENYENT)(long dividend)\
     {\
          coset = dividend;\
          simplify();\
     }\
     mnum(ENYENT)(const mnum(ENYENT)& a) \
     { \
          coset = a.coset; \
          simplify(); \
     } \
     mnum(ENYENT)() \
     {  \
        \
     }  \
     void assign(INTEGER dividend) \
     {  \
        coset = dividend; \
        simplify(); \
     }  \
     mnum(ENYENT)& operator= (const mnum(ENYENT) &num) \
     {  \
        coset = num.coset;  \
        return *this; \
     }  \
     mnum(ENYENT)& operator+= (const mnum(ENYENT) &num) \
     {  \
          coset += num.coset; \
          simplify(); \
          return *this; \
     }  \
     mnum(ENYENT)& operator-= (const mnum(ENYENT) &num) \
     {  \
          coset -= num.coset; \
          simplify(); \
          return *this; \
     }  \
     mnum(ENYENT)& operator*= (const mnum(ENYENT) &num) \
     {  \
          coset *= num.coset; \
          simplify(); \
          return *this; \
     }  \
     static mnum(ENYENT) unity()\
     {\
        mnum(ENYENT) num = 1;\
        return num;\
     }\
     static mnum(ENYENT) zero()\
     {\
     mnum(ENYENT) num = 0;\
        return num;\
     }\
     static void error(char*);\
     INTEGER var() const\
     {\
          return coset;\
     }\
         INTEGER dclass() const\
     {\
          return divisor;\
     }\
     mnum(ENYENT) operator-() const\
     {\
        mnum(ENYENT) num;\
        num = zero() - *this;\
        return num;\
     }\
     mnum(ENYENT) inverse() const\
     { \
          mnum(ENYENT) num = *this; \
          num.simplify(); \
          if  (num.coset == 0)  error("zero has no inverse"); \
          if  (gcd(divisor,coset) != 1) \
             error("attempting to find inverse of a number with no inv."); \
          for ( int i = 1; i < divisor; i++) \
          { \
             if ( ( (i*coset-1) % divisor) == 0) break; \
          } \
          mnum(ENYENT) temp(i); \
          return temp; \
     } \
     mnum(ENYENT)& operator/= (const mnum(ENYENT) &num) \
     { \
          mnum(ENYENT) temp = num.inverse(); \
          coset *= temp.coset; \
            simplify(); \
            return *this; \
        } \
        mnum(ENYENT) operator+(const mnum(ENYENT)&) const;\
        mnum(ENYENT) operator-(const mnum(ENYENT)&) const;\
        mnum(ENYENT) operator*(const mnum(ENYENT)&) const;\
        mnum(ENYENT) operator/(const mnum(ENYENT)&) const;\
        int operator==(const mnum(ENYENT)&) const; \
        int operator!=(const mnum(ENYENT)&) const;\
        friend ostream& operator<<(ostream &s, const mnum(ENYENT)&); \
        friend istream& operator>>(istream &s, mnum(ENYENT)&); \
        friend mnum(ENYENT) dotprod(mnum(ENYENT) *x, mnum(ENYENT) *y, int n); \
        friend void matmult(mnum(ENYENT) *rows, mnum(ENYENT) *vector1, \
              mnum(ENYENT) *vector2, int rowdim, int coldim); \
};

#endif GENERIC_MOD_CLASS_H
```

7.5.1.2 File gen_mod2.cpp

```
// implementation of class generic modulo

#include "gen_mod2.h"

#define moduloimplement(ENYENT) \
INTEGER mnum(ENYENT)::divisor = ENYENT;\
mnum(ENYENT) dotprod(mnum(ENYENT)* x, mnum(ENYENT)* y, int dim) { \
    mnum(ENYENT) result(0); \
    for (int i = 0; i < dim; i++) result += x[i]*y[i]; \
    return result; \
} void matmult(mnum(ENYENT) *rows, mnum(ENYENT) *vector1, \
      mnum(ENYENT) *vector2, int rowdim, int coldim) \
{ \
     mnum(ENYENT) *prow, dotprod(mnum(ENYENT)*, mnum(ENYENT)*, int); \
     for (int i = 0; i < rowdim; ++i) \
     { \
         prow = rows + i*coldim; \
         vector2[i] = dotprod(prow, vector1, coldim); \
     } \
} INTEGER mnum(ENYENT)::LABS(INTEGER number) {\
   return (( number > 0) ? number : -number);}\
INTEGER mnum(ENYENT)::gcd(INTEGER dividend, INTEGER divisor)\
{\
    dividend  = LABS(dividend);\
    divisor = LABS(divisor);\
    if ( (divisor == 1) || (dividend == 1) ) return 1;\
    if (divisor > dividend)\
      {\
     INTEGER temp = dividend;\
     dividend = divisor;\
     divisor = temp;\
    }\
    if (divisor == 0) return(dividend ? dividend : 1 );\
      INTEGER rem = dividend - (dividend/divisor)*divisor;\
    while (rem != 0)\
    {\
     dividend = divisor;\
     divisor = rem;\
     rem = dividend - (dividend/divisor)*divisor;\
    }\
    return(divisor);\
}\
void mnum(ENYENT)::error(char* p)\
{\
    cerr << p << "\n";\
    exit(1);\
}\
ostream& operator<< (ostream &s,const mnum(ENYENT) &num) \
{ \
     return(s<<num.coset<<" mod " << num.divisor); \
} \
istream& operator>> (istream &s, mnum(ENYENT) &num) \
{ \
    long n; \
    s>> n; \
    num = mnum(ENYENT)(n); \
    return s; \
} \
mnum(ENYENT) mnum(ENYENT)::operator+(const mnum(ENYENT) &num) const\
{ \
   return mnum(ENYENT)(coset+num.coset); \
} \
mnum(ENYENT) mnum(ENYENT)::operator-(const mnum(ENYENT) &num) const\
{ \
   return mnum(ENYENT)(coset-num.coset); \
} \
mnum(ENYENT) mnum(ENYENT)::operator*(const mnum(ENYENT) &num) const\
{ \
   return mnum(ENYENT)(coset*num.coset); \
} \
mnum(ENYENT) mnum(ENYENT)::operator/(const mnum(ENYENT)& num) const\
{ \
   mnum(ENYENT) temp = num.inverse(); \
   return mnum(ENYENT)(coset*temp.coset); \
} \
int mnum(ENYENT)::operator==(const mnum(ENYENT) &num) const\
{ \
     mnum(ENYENT) temp1 = *this;\
     temp1.simplify(); \
     mnum(ENYENT) temp2 = num;\
     temp2.simplify();  \
     return  (temp1.coset==temp2.coset) ; \
} \
int mnum(ENYENT)::operator!=(const mnum(ENYENT) &num) const\
{ \
     mnum(ENYENT) temp1 = *this;\
     temp1.simplify(); \
       mnum(ENYENT) temp2 = num;\
     temp2.simplify(); \
     return (temp1.coset != temp2.coset) ; \
}


// EXPAND THE MACROS

declare(modulo,5);
implement(modulo,5);
```

7.5.1.3 File prgvec1.h

```
#define MNUM5
// #define DOUBLE
// #define COMPLEX
// #define MODULO_NUMBER
// #define RATIONAL_NUMBER
#pragma option -Jgd
#ifdef  MNUM5
   #include <generic.h>
     #include "gen_mod2.h"
     declare(modulo,5);
     typedef vector<mnum5> fake_vec;
     typedef matrix<mnum5> fake_mat;
   typedef vector<mnum5> fake_vec;
   typedef matrix<mnum5> fake_mat;
#endif

#ifdef DOUBLE
   #include "double.h"
     typedef vector<Double> fake_Dou_vec;
   typedef matrix<Double> fake_Dou_mat;
#endif

#ifdef COMPLEX
     #include "complex.h"
   typedef vector<Complex> fake_Com_vec;
   typedef matrix<Complex> fake_Com_mat;
#endif
```

```
#ifdef RATIONAL_NUMBER
   #include "rational.h"
     typedef vector<rational_number> fake_rat_vec;
     typedef matrix<rational_number> fake_rat_mat;
#endif

#ifdef MODULO_NUMBER
     #include "modulo.h"
     typedef vector<modulo_number> fake_rat_vec;
     typedef matrix<modulo_number> fake_rat_mat;
#endif
```

7.5.1.4 File genmtest.cpp

```
#include <iostream.h>
#pragma option -Jgx
#include <generic.h>
#include <stdlib.h>

#pragma option -Jgx
#include "gen_mod2.h"
#include "vector.h"

declare(modulo,5);
vector<mnum5> operator*(const mnum5& n,
     const vector<mnum5> &tv);
matrix<mnum5> operator*(const mnum5& n,
     const matrix<mnum5> &tv);
ostream& operator<<(ostream& s, const matrix<mnum5> &tm);
ostream& operator<<(ostream& s, const vector<mnum5> &tm);
istream& operator>>(istream& s, const matrix<mnum5> &tm);
istream& operator>>(istream& s, const vector<mnum5> &tv);

int main() {
      matrix<mnum5> rm(3,4),am(3,4),bm(3,4),cm(3,4),dm(3,4),em(3,4),fm(3,4),
                       gm(3,4);
      mnum5 v[3], ww[3] ,y[3],zz[3],  aa, bb;
      v[0].assign(0); v[1].assign(3); v[2].assign(7);
      ww[0].assign(2); ww[1].assign(1); ww[2].assign(3);
      zz[0].assign(2); zz[1].assign(2); zz[2].assign(4);
      mnum5 x[9]  ;
      matrix<mnum5> q(3,3);
      vector<mnum5> uu(3), tt(3) ;   /*  B 4.0 */
      uu[0] = v[0]; uu[1] = v[1]; uu[2] = v[2];
      tt =  mnum5(2) * uu;
    rm.elem(0,0) = v[0]; rm.elem(0,1) = v[1]; rm.elem(0,2) = v[2];
    rm.elem(1,0) = ww[0]; rm.elem(1,1) = ww[1]; rm.elem(1,2) = ww[2];
    rm.elem(2,0) = zz[0]; rm.elem(2,1) = zz[1]; rm.elem(2,2) = zz[2];
    rm.elem(0,3) = ww[2]; rm.elem(1,3) = zz[1]; rm.elem(2,3) = ww[0];
    matrix<mnum5> gjrm = rm.gauss_jordan();
    cout << "\nrm is " << rm;
    cout << "\ngjrm is " << gjrm;
    am = bm = rm;
    cout << "\nam is " << am;
    cout << "\nbm is " << bm;
    cout << "\nrm is " << rm;
    rm += am;
    cout << "\nam is " << am;
    cout << "\nbm is " << bm;
    cout << "\nrm is " << rm;
    em = rm - gjrm;
    cout << "\nem is " << em;
      rm -= gjrm;
    cout << "\nrm is " << rm;
  /*  cm = 2*rm;  */
      cm = rm*mnum5(2);    /*  Borland 4.0 */
    cout << "\ncm is " << cm;
    rm *= 2;
    cout << "\nrm is " << rm;
    dm = rm + cm;
      cout << "\ndm is " << dm;
    rm += cm;
    cout << "\nrm is " << rm;
    fm = -cm;
    gm = -cm;
    cout << "\nfm is " << fm;
    cout << "\ngm is " << gm;
    if (fm != gm) cout << "\nfm and gm are not equal" ;
    else cout << "\nfm and gm are equal";
    fm[2][2]=mnum5(1);
    cout << "\n fm had a component change";
    if (fm != gm) cout << "\nfm and gm are not equal" ;
    else cout << "\nfm and gm are equal";
    cout << "\nfm is " << fm;
    cout << "\ngm is " << gm;
    return 1;
}
```