Overloading Assignment

- Using the assignment symbol

  a2=a1;

  causes the compiler to copy the data from a1, member by member into a2. This is the default action of the assignment operator =.
- However, there might be situations in which you want the assignment operator to behave differently, for example if your data member is a pointer to objects you might want to replicate the objects itself as well not just the pointers.

Linked List Example

```
struct link      // one element of list
{
   int data;    // data item
   link *next;  // pointer to next element
};

class linklist
{
   private:
      link* first;    // pointer to first link
   public:
      linklist() { first = NULL;}    // no argument constructor
      void additem(int d);        // add data item (one link)
      void display();          // display all links
}
```

Linked List Example

```
void linklist::additem(int d)   // add data item
{
    link* newlink = new link;   // create a new link
    newlink->data = d;          // give it data d
    newlink->next=first;        // it points to the next link
    first = newlink;            // now first points to this link
}
void linklist::display()    // display all links
{
   link* current=first;    // set ptr to first link
   while(current != NULL)   // until ptr points beyond last link
   {
      cout << current->data << " ";   // print data
       current=current->next;          // move to next link
   }
}
```

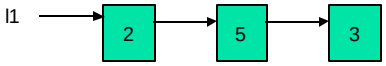
Linked List Example

```
void linklist::deleteitem()   // delete first data item
{
    link* tmp=first->next;   // tmp to remember pointer to 2nd element
    delete first;  // deletes first link from memory
    first=tmp;    // old second element becomes new first element
}
```

Linked List Example

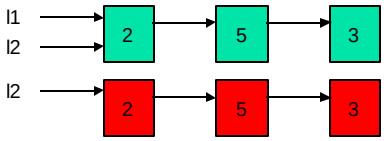
- Assume you assign one list to another, with the default assignment operator only the pointer to the first link gets copied.

```
linklist l1;
l1.additem(3);
l1.additem(5);
l1.additem(2);
```



Linked List Example

```
linklist l1;
l1.additem(3);
l1.additem(5);
l1.additem(2);
linklist l2;
l2=l1;
```



Overloading Assignment

```
linklist& linklist::operator=(linklist &list)   // assignment operator
{
   while (first !=NULL)   // first empty list
      deleteitem();

   link* current=list.first;    // set ptr to first link
   while(current != NULL)   // until ptr points beyond last link
   {
        additem(current->data);   // print data
        current=current->next;      // move to next link
   }
   first=current;
   return *this;
 }
```

Murphy's Law

```
linklist l1;
l1.additem(3);
l1.additem(5);
l1.additem(2);
l1=l1;   //  oooouuuch !!!!  l1 deletes itself

linklist& linklist::operator=(linklist &list)   // assignment operator
{
   if (this == &list)    // both arguments to =  are the same  object
      return *this;
   ...
}
```

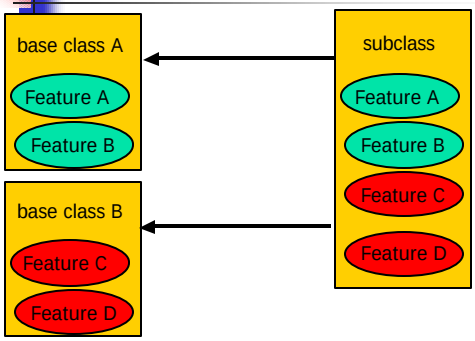
Copy Constructor

- You can define and at the same time initialize an object to a value of another object with two kinds of statements.

```
linklist l1(l2);      // copy initialization
linklist l1=l2;     // copy initialization  not assignment
l1=l2;                  // assignment operator =

class linklist
{
 public:
     linklist() { first = NULL;}
     linklist( linklist& list) { *this=list; }    // copy constructor
     ...
 };
```

Multiple Inheritance



Multiple Inheritance

```
class Date
{
    private:
      int day, month, year;
    …
};
class Time
{
    private:
      int hours, minutes;
    …
};
```

Multiple Inheritance

```
class DateTime : public Date, public Time
{
    public:
      DateTime(int d, int m, int y, int h, int mi)
         : Date(d,m,y), Time(h, mi)  {};
    …
};
```

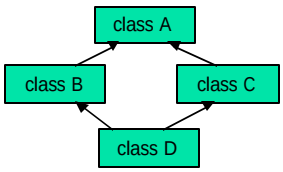
Ambiguity in Multiple Inheritance

```
class Date
{
    void add(int days);
};
class Time
{
   void add(int minutes);
};
DateTime dt(13,2,1998,23,10);
dt.add(3);   // ambiguous -- will not compile
dt.Date::add(4);  // uses add of class Date
dt.Time::add(5);  // uses add of class Time
```

Ambiguity in Multiple Inheritance

```
class A {  public:  void F(); };
class B : public A { …};
class C : public A { …};
class D : public B, public C {};
D d;
d.F();   // ambiguous - won´t compile
```

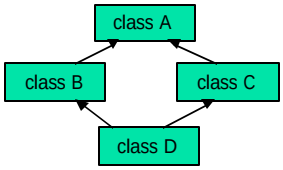
diamond shaped inheritance tree



Ambiguity in Multiple Inheritance

```
class A {  public:  void F(); };
class B : virtual public A { …};
class C : virtual public A { …};
class D : public B, public C {};
D d;
d.F();   // OK – only one copy of class A object
```

diamond shaped inheritance tree



Streams and Files

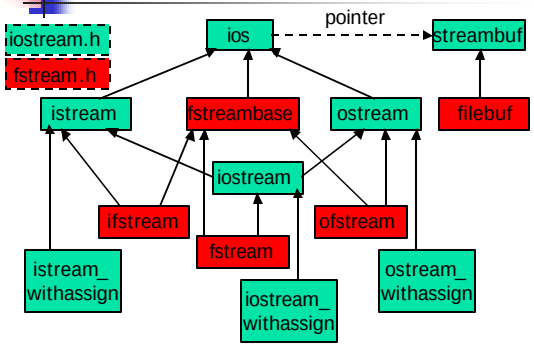
- A stream is a flow of data. In C++ streams are represented by objects of the class ios and its subclasses.
- Different types of streams represent different kinds of data flow, for example the class ifstream represents data flow from input disk files.
- The advantage of streams compared to traditional C-functions such as printf(), scanf() is that each object already knows how to display itself.

```
int a;
float x;
printf("integer %d, float %f\n",a,x);
cout << "integer " << a << ", float "<< x << endl;
```

Streams and Files

- Another advantage of streams in C++ is that you can overload the operators insertion <<, and extraction >> operators to work with your own classes.
- Even if your program uses a graphical user interface library such as grain, Xwindows or MS libraries, streams are still needed for file input/output.

Stream Class Hierarchy



Stream Class Hierarchy

- ios : is the base class that contains constants and member functions common to input and output operations. It also contains a pointer to streambuf which contains the actual memory buffer into which data is read or written.
- istream, ostream, iostream : are derived from ios, are dedicated to input and output and contain functions such as
  - istream: get(), getline(), read() and the >> operator
  - ostream: put(), write() and the << operator
  - iostream inherits from both istream and ostream

Formatting Flags

- Formatting flags act as switches that specify choices for various aspects of input and output
  - left : left adjust output [12.4]
  - right : right adjust output [12.4]
  - dec, hex, oct : decimal, octal, hexadecimal conversion
  - fixed, scientific : use fixed, scientific format on floating point output

```
cout.setf(ios::left);  // left justified output
cout.unsetf(ios::left);  // return to default (right justified)
```

Manipulators

- Manipulators are formatting instructions directly inserted into a stream.
  - endl : Inserts newline and flush output stream
  - flush : flush output stream
  - lock, unlock : lock, unlock file handle
  - setw(int) : set field width of output
  - setfill(int) : set fill character for output
  - setprecision(int) : set precision (# of digits displayed)
  - setiosflags(long), resetiosflags(long) : set/reset specified flags

```
#include <iomanip>
float x=3.14259;
cout << "[" << setw(8) << setprecision(4) << setiosflags(ios::left)
        << setfill('*') << x << "]" << endl;    // displays   [3.143***]
```

Functions

- The ios class contains a number of functions that can be used to set formatting flags.
  - fill(int) : sets fill character
  - precision(int) : sets the precision
  - width(int) : sets the width of output
  - setf(long) : set flags

```
cout.width(5);
cout.precision(6);
cout.setf(ios::left);
```

Istream Class

- The istream class, derived from ios, performs input-specific activities or extraction.
  - >> : extraction operator
  - get(ch); : extract one character into ch
  - get(str); : extract characters into array str, until '\0'
  - putback(ch); : insert last character back into input stream
  - read(str, MAX) : (for files) extract up to MAX characters into str until EOF

```
char ch='n';
while(ch != 'y') {
  cout << "Enter y : " << endl;
   cin.get(ch);   }
```

Ostream Class

- The ostream class, derived from ios, performs output-specific activities or extraction.
  - << : insertion operator
  - put(ch); : insert one character into ch
  - flush(); : flush buffer contents and insert new line
  - write(str, SIZE) : (for files) insert SIZE characters from str into file

```
cout.put('a');
cout.flush();
```

Iostream with Assign

- istream_withassign, ostream_withassign, iostream_withassign are identical to istream, ostream and iostream except that stream objects can be copied.
- Normally the stream objects can not be copied as they point to the same streambuf object

Stream Errors

- The stream error-status flags report errors that occur in input or output operations.
- Various ios functions can be used to read and set these error flags.
  - eofbit : reached end of file
  - goodbit : no errors
  - failbit : operation failed
  - badbit : invalid operation (no associated streambuf)
  - hardfail : unrecoverable error
- Functions for error flags
  - int = eof(); : returns true if EOF flag set
  - int = good(); : returns true if OK

Stream Errors

```
cout << "enter an integer: ";
cin >> i;
while (! cin.good())
 {
    cin.clear();  // clear the error bits
    cin.ignore(10, '\n');   // remove newline from cin
    cout << "incorrect input enter an integer: ";
    cin >> i;
 }
```

Disk File I/O with Streams

- ifstream : input from files
- ofstream : output to files
- fstream : both input and output to files
- declared in header file <fstream>

```
#include <fstream>
int n=12;
string str= "Shakespeare";
ofstream outfile("data.txt");   // create ofstream object
outfile << n << " th Night was written by " << str << endl;
outfile.close();  // explicitly closes the file
```

Disk I/O with File Streams

```
#include <fstream>
const int BUF_SIZE = 80;
char buffer[BUF_SIZE];  // character buffer
ifstream infile("test.txt");
while ( !infile.eof() )    // until end of file
{
   infile.getline(buffer,BUF_SIZE);   // read a line of text
    cout << buffer << endl;  // display it
}
```

Character I/O

- The put() and get() functions can be used to output and input single characters.

```
#include <fstream>
#include <string>
string str="Love sees not with the eye but with the mind
   and therefore Cupid's wings are painted blind";
ofstream outfile("text.txt");  // create file for output
for (int i=0; i<str.size(); i++) // for each character
   outfile.put( str[i] );  // write char to file
outfile.close();
```

Character I/O

```
#include <fstream>
char ch;
ifstream infile("text.txt");  // create file for output
while ( ! infile.eof() )   // read until end of file for (int i=0; {
    {
     infile.get(ch);  // read character
     cout < ch;  // display it
    }
infile.close();
```

Mode Bits for Opening Files

- The open() function can be used to open a file:
- The mode bits specify how the stream object is opened
  - in : open for reading
  - out : open for writing
  - app : start writing at end of file (append)
  - trunc : truncate file to zero length if exists
  - nocreate : error when opening if file does not exist
  - noreplace : error when opening if file does exist

```
fstream outfile;   // defines the fstream variable
outfile.open("test.data", ios::app);
   // opens the file in append mode
```

Overloading << Operator

- Overloading the << operator allows you to specify the way in which an object is displayed
- As the << operator expects a stream object as the left hand argument it must be overloaded as a non-member function :

ostream& operator<<(ostream& os, Date d);

- It is possible to grant the non-member function access to the private data member of the class by declaring the function as a *friend* of the class.

Overloading << Operator

```
#include <fstream>
class Date
{
   friend ostream& operator<<(ostream& os, Date d);
};

ostream& operator<<(ostream& os, Date d)
{
   os << d.day << "." << d.month << "." << d.year;
   //access private data as friend
};

Date d1(16,3,1998);
ofstream datefile("dates.txt");
datefile <<   d1  << endl;
```

Exceptions

- Exceptions provide a systematic, object-oriented approach to handle run-time errors generated by C++ classes.
- Exceptions are errors that occur at run-time, for example caused by running out of memory, not being able to open a file or using out-of-bounds index to a vector.
- In C errors are usually signaled by the return status of a function. The drawback is that after each call to a function it is necessary to examine the return value, which requires a lot of code and makes the listing hard to read.

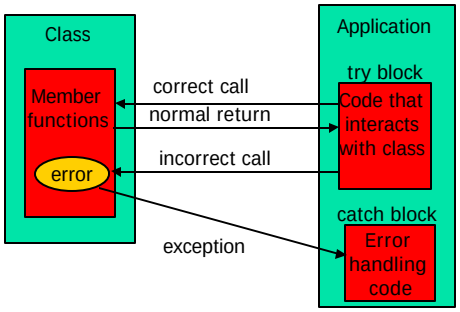
Exceptions

- Imagine an application that creates and interacts with objects of a certain class.
- When the application causes an error in one of the member functions and exception is *thrown*.
- The code that handles the exception is an *exception handler* or *catch block*.
- Any code in the application that uses objects of the class is enclosed in a *try* block.
- Errors generated in the try block will be caught in the catch block.

Exceptions

```
class frac
{
    private:
      int num, den;
    public:
      class fracerror {};  // exception class
      frac( double n, double d)   // constructor
      {
        if ( d == 0)   // error condition
          throw fracerror;  // throw exception
        else { num=n; den=d; }
      }
};
```

Exception Mechanism



Exceptions

```
int d,n;
char dummy;
while (d==0) {
    try  // try block
    {
          cout << "enter a/b";
          cin >> n >> dummy >> d;
          frac f(n,d);   // calls constructor for frac
    }
    catch(frac::fracerror)    // exception handler
    {
          cout << "denumerator 0 not allowed" << endl;
    }
}
```

Exceptions

```
class Vec
{
    private:
      int size;
      double *array;
    public:
      class Range {}; // exception class for Vec
      double operator[] ( int index)
      {
        if ((index <0) || ( index >= size))
          throw Range();
        else return array[index];
      }
};
```

Exceptions

```
Vec x;
try {       // try block
    x.push_back(12.0);
    x.push_back(3.0);
    x.push_back(5.0);
    x.push_back(2.3);
    cout << x[5] << endl;   // ooops, no 5th element
}
catch ( Vec::Range)   // exception handler
{
    cout << " Index out of range " << endl;
}
```