**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**

```cpp
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
};

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
    }
}

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

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

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

```cpp
linklist l2;
l2 = l1;
```

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

```cpp
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;
}
```

```cpp
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

```
class Date
{
  private:
    int day, month, year;
    ...  
};

class Time
{
  private:
    int hours, minutes;
    ...  
};

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

```cpp
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

```cpp
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
```

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.

```cpp
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 X11, Xwindows or MS libraries, streams are still needed for file input/output.

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.
  - \texttt{endl} : Inserts newline and flush output stream
  - \texttt{flush} : flush output stream
  - \texttt{lock, unlock} : lock, unlock file handle
  - \texttt{setw(int)} : set field width of output
  - \texttt{setfill(int)} : set fill character for output
  - \texttt{setprecision(int)} : set precision (# of digits displayed)
  - \texttt{setiosflags(long), resetiosflags(long)} : set/reset specified flags

\#include \texttt{\textless iomanip}\textgreater
float x=3.14259;
cout << "\[" << \texttt{setw(8)} << \texttt{setprecision(4)} << \texttt{setiosflags(ios::left)}
    << \texttt{setfill('*')} << x << "\]" << \texttt{endl};  // displays \[3.143***\]

Functions

- The \texttt{ios} class contains a number of functions that can be used to set formatting flags.
  - \texttt{fill(int)} : sets fill character
  - \texttt{precision(int)} : sets the precision
  - \texttt{width(int)} : sets the width of output
  - \texttt{setf(long)} : set flags
    \texttt{cout.width(5)};
    \texttt{cout.precision(6)};
    \texttt{cout.setf(ios::left)};

Istream Class

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

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

Ostream Class

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

\texttt{cout.put(\textasciitilde\textgreater\textasciitilde)};
\texttt{cout.flush()};
Iiostream 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

Disk File I/O with Streams

```cpp
#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

```cpp
#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

```cpp
#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();
```

### Mode Bits for Opening Files

- The open() function can be used to open a file:
  - 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

```cpp
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:

```cpp
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.

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

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

```cpp
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.

- 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

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

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;  // oops, no 5th element
} catch (Vec::Range) {  // exception handler
    cout << "Index out of range" << endl;
}