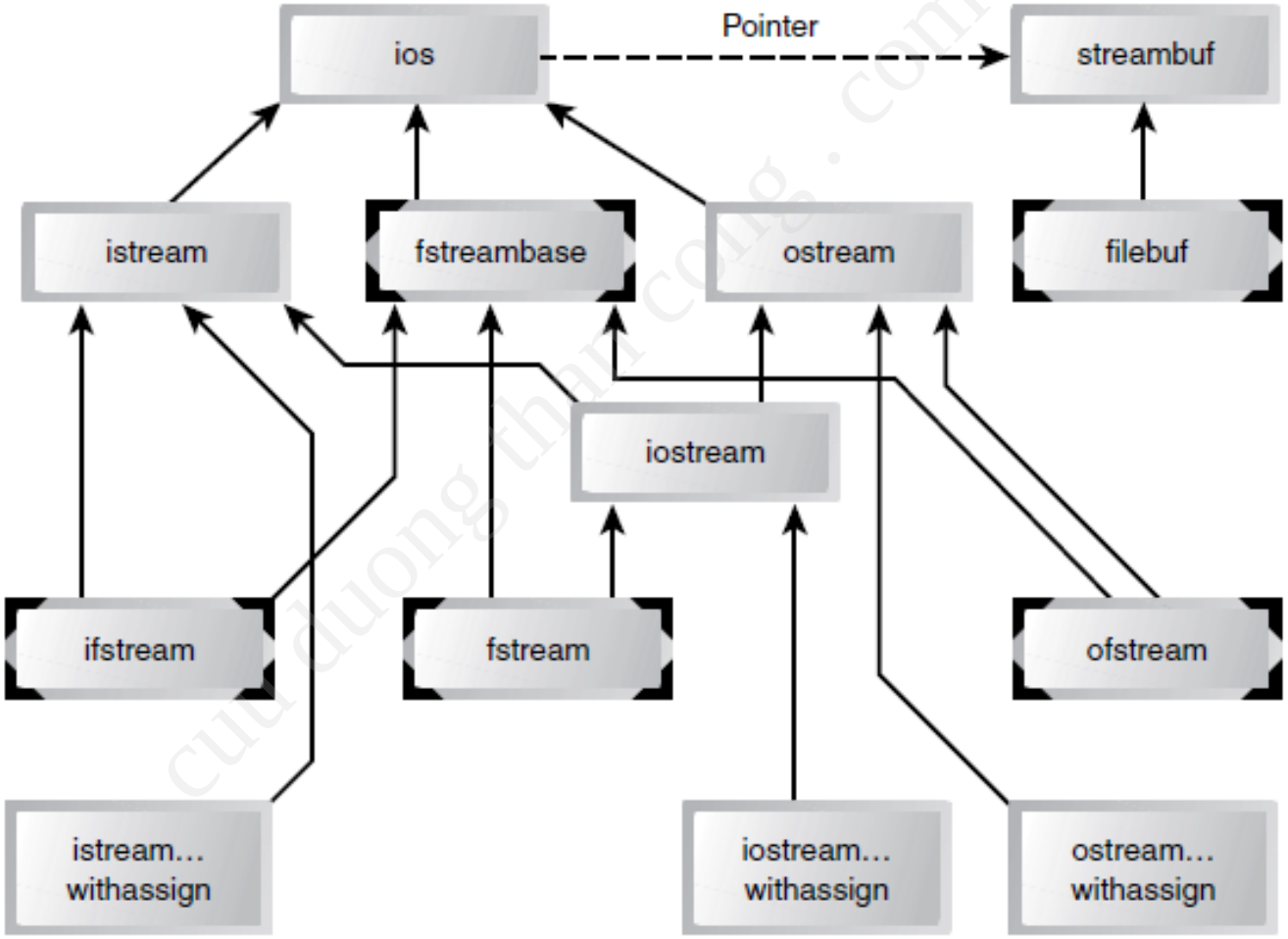
Streams and Files

Ho Dac Hung

Stream Classes

- A stream is a general name given to a flow of data. In C++ a stream is represented by an object of a particular class. So far we've used the cin and cout stream objects. Different streams are used to represent different kinds of data flow.

The Stream Class Hierarchy



The ios Class

- The ios class is the granddaddy of all the stream classes, and contains the majority of the features you need to operate C++ streams. The three most important features are the formatting flags, the error-status flags, and the file operation mode.

Formatting Flags

- Formatting flags are a set of enum definitions in ios. They act as on/off switches that specify choices for various aspects of input and output format and operation.
- All the flags can be set using the setf() and unsetf() ios member functions.

```
cout.setf(ios::left); // left justify output text
cout >> “This text is left-justified”;
cout.unsetf(ios::left); // return to default (right
        justified)
```

Formatting Flags

Flag	Meaning
skipws	Skip (ignore) whitespace on input
left	Left-adjust output [12.34]
right	Right-adjust output [12.34]
internal	Use padding between sign or base indicator and number [+ 12.34]
dec	Convert to decimal
oct	Convert to octal
hex	Convert to hexadecimal
boolalpha	Convert bool to “true” or “false” strings
showbase	Use base indicator on output (0 for octal, 0x for hex)
showpoint	Show decimal point on output

Formatting Flags

Flag	Meaning
uppercase	Use uppercase X, E, and hex output letters (ABCDEF)—the default is lowercase
showpos	Display + before positive integers
scientific	Use exponential format on floating-point output [9.1234E2]
fixed	Use fixed format on floating-point output [912.34]
unitbuf	Flush all streams after insertion
stdio	Flush stdout, stderror after insertion

Manipulators

- Manipulators are formatting instructions inserted directly into a stream.

```
cout << setiosflags(ios::fixed)
<< setiosflags(ios::showpoint)
<< var;
```

Manipulators

Manipulator	Purpose
ws	Turn on whitespace skipping on input
dec	Convert to decimal
oct	Convert to octal
hex	Convert to hexadecimal
endl	Insert newline and flush the output stream
ends	Insert null character to terminate an output string
flush	Flush the output stream
lock	Lock file handle
unlock	Unlock file handle

Manipulators

Manipulator	Argument	Purpose
setw()	field width (int)	Set field width for output
setfill()	fill character (int)	Set fill character for output
setprecision()	precision (int)	Set precision
setiosflags()	formatting flags (long)	Set specified flags
resetiosflags()	formatting flags (long)	Clear specified flags

Functions

- The ios class contains a number of functions that you can use to set the formatting flags and perform other tasks.

```
cout.width(14);
cout.fill('*');
cout.setf(ios::left);
cout.unsetf(ios::left);
```

Functions

Manipulator	Purpose
ch = fill();	Return the fill character
fill(ch);	Set the fill character
p = precision();	Get the precision
precision(p);	Set the precision
w = width();	Get the current field width
width(w);	Set the current field width
setf(flags);	Set specified formatting flags
unsetf(flags);	Unset specified formatting flags
setf(flags, field);	First clear field, then set flags

The istream Class

- The istream class, which is derived from ios, performs input-specific activities, or extraction. It's easy to confuse extraction and the related output activity, insertion.

The istream Class

Function	Purpose
>>	Formatted extraction for all basic (and overloaded) types.
get(ch);	Extract one character into ch.
get(str)	Extract characters into array str, until '\n'.
get(str, MAX)	Extract up to MAX characters into array.
get(str, DELIM)	Extract characters into array str until specified delimiter.
get(str, MAX, DELIM)	Extract characters into array str until MAX characters or the DELIM character.
getline(str, MAX, DELIM)	Extract characters into array str, until MAX characters or the DELIM character.
putback(ch)	Insert last character read back into input stream.
ignore(MAX, DELIM)	Extract and discard up to MAX characters until the specified delimiter

The istream Class

Function	Purpose
peek(ch)	Read one character, leave it in stream.
count = gcount()	Return number of characters read by a call to get(), getline(), or read().
read(str, MAX)	For files—extract up to MAX characters into str, until EOF.
seekg()	Set distance (in bytes) of file pointer from start of file.
seekg(pos, seek_dir)	Set distance (in bytes) of file pointer from specified place in file. seek_dir can be ios::beg, ios::cur, ios::end.
pos = tellg(pos)	Return position (in bytes) of file pointer from start of file.

The ostream Class

- The ostream class handles output or insertion activities.

The ostream Class

Function	Purpose
<<	Formatted insertion for all basic (and overloaded) types.
put(ch)	Insert character ch into stream.
flush()	Flush buffer contents and insert newline.
write(str, SIZE)	Insert SIZE characters from array str into file.
seekp(position)	Set distance in bytes of file pointer from start of file.
seekp(position, seek_dir)	Set distance in bytes of file pointer, from specified place in file. seek_dir can be ios::beg, ios::cur, or ios::end.
pos = tellp()	Return position of file pointer, in bytes.

Error-Status Bits

- The stream error-status flags constitute an ios enum member that reports errors that occurred in an input or output operation.
- Various ios functions can be used to read (and even set) these error flags.

Error-Status Bits

Name	Meaning
goodbit	No errors (no flags set, value = 0)
eofbit	Reached end of file
failbit	Operation failed (user error, premature EOF)
badbit	Invalid operation (no associated streambuf)
hardfail	Unrecoverable error

Error-Status Bits

Function	Purpose
int = eof();	Returns true if EOF flag set
int = fail();	Returns true if failbit or badbit or hardfail flag set
int = bad();	Returns true if badbit or hardfail flag set
int = good();	Returns true if everything OK; no flags set
clear(int=0);	With no argument, clears all error bits; otherwise sets specified flags, as in clear(ios::failbit)

```
while(true) // cycle until input OK
{
    cout << "\nEnter an integer: ";
    cin >> i;
    if( cin.good() ) // if no errors
    {
        cin.ignore(10, '\n'); // remove newline
        break; // exit loop
    }
    cin.clear(); // clear the error bits
    cout << "Incorrect input";
    cin.ignore(10, '\n'); // remove newline
}
cout << "integer is " << i; // error-free integer
```

Disk File I/O with Streams

- Most programs need to save data to disk files and read it back in. Working with disk files requires another set of classes: ifstream for input, fstream for both input and output, and ofstream for output. Objects of these classes can be associated with disk files, and we can use their member functions to read and write to the files.

```
int main()
{
    char ch = ‘x’;
    int j = 77;
    double d = 6.02;
    string str1 = “Kafka”; //strings without
    string str2 = “Proust”; // embedded spaces
    ofstream outfile(“fdata.txt”); //create ofstream object
    outfile << ch //insert (write) data
        << j
        << ‘ ‘ //needs space between numbers
        << d
        << str1
        << ‘ ‘ //needs spaces between strings
        << str2;
        cout << “File written\n”;
    return 0;
}
```

```
int main()
{
    char ch;
    int j;
    double d;
    string str1;
    string str2;
    ifstream infile(“fdata.txt”); //create ifstream object
    infile >> ch >> j >> d >> str1 >> str2;
    cout << ch << endl //display the data
        << j << endl
        << d << endl
        << str1 << endl
        << str2 << endl;
    return 0;
}
```

```
int main()
{
    ofstream outfile(“TEST.TXT”);
    outfile << “I fear thee, ancient Mariner!\n”;
    outfile << “I fear thy skinny hand\n”;
    outfile << “And thou art long, and lank, and
        brown,\n”;
    outfile << “As is the ribbed sea sand.\n”;
    return 0;
}
```

```
int main()
{
    const int MAX = 80; //size of buffer
    char buffer[MAX]; //character buffer
    ifstream infile("TEST.TXT"); //create file for input
    while( !infile.eof() ) //until end-of-file
    {
        infile.getline(buffer, MAX); //read a line of text
        cout << buffer << endl; //display it
    }
    return 0;
}
```

Character I/O

- The put() and get() functions, which are members of ostream and istream, respectively, can be used to output and input single characters.

Character I/O

```
int main()
{
    string str = “Time is a great teacher, but
         unfortunately it kills all its pupils. Berlioz”;
    ofstream outfile(“TEST.TXT”);
    for(int j=0; j<str.size(); j++) //for each character,
    outfile.put( str[j] ); //write it to file
    cout << “File written\n”;
    return 0;
}
```

Character I/O

```
int main()
{
    char ch; //character to read
    ifstream infile(“TEST.TXT”); //create file for input
    while( infile ) //read until EOF or error
    {
        infile.get(ch); //read character
        cout << ch; //display it
    }
    cout << endl;
    return 0;
}
```

Character I/O

```
int main()
{
    ifstream infile(“TEST.TXT”); //create file for input
    cout << infile.rdbuf(); //send its buffer to cout
    cout << endl;
    return 0;
}
```

Binary I/O

- You can write a few numbers to disk using formatted I/O, but if you're storing a large amount of numerical data it's more efficient to use binary I/O, in which numbers are stored as they are in the computer's RAM memory, rather than as strings of characters.

```
int main()
{
    for(int j=0; j<MAX; j++) //fill buffer with data
    buff[j] = j; //(0, 1, 2, ...)
    ofstream os(“edata.dat”, ios::binary);
    os.write( reinterpret_cast<char*>(buff), MAX*sizeof(int) );
    os.close(); //must close it
    for(j=0; j<MAX; j++) //erase buffer
    buff[j] = 0;
    ifstream is(“edata.dat”, ios::binary);
    is.read( reinterpret_cast<char*>(buff), MAX*sizeof(int) );
    for(j=0; j<MAX; j++) //check data
    if( buff[j] != j )
    { cerr << “Data is incorrect\n”; return 1; }
    cout << “Data is correct\n”;
    return 0;
}
```

Object I/O

- Since C++ is an object-oriented language, it's reasonable to wonder how objects can be written to and read from disk.

Object I/O

```
class person //class of persons
{
    protected:
        char name[80]; //person's name
        short age; //person's age
    public:
        void getData() //get person's data
        {
            cout << "Enter name: "; cin >> name;
            cout << "Enter age: "; cin >> age;
        }
};
```

Object I/O

```
int main()
{
    person pers; //create a person
    pers.getData(); //get data for person
    ofstream outfile("PERSON.DAT", ios::binary);
    outfile.write(reinterpret_cast<char*>(&pers),
        sizeof(pers));
    return 0;
}
```

Object I/O

```
int main()
{
    person pers; //create person variable
    ifstream infile(“PERSON.DAT”, ios::binary);
    infile.read(          reinterpret_cast<char*>(&pers),
        sizeof(pers) );
    pers.showData(); //display person
    return 0;
}
```

The fstream Class

- To create a file that can be used for both input and output, we need an object of the fstream class, which is derived from iostream, which is derived from both istream and ostream so it can handle both input and output.
- We create the file in one statement and open it in another, using the open() function, which is a member of the fstream class.

The fstream Class

Mode Bit	Result
in	Open for reading (default for ifstream)
out	Open for writing (default for ofstream)
ate	Start reading or writing at end of file (AT End)
app	Start writing at end of file (APPend)
trunc	Truncate file to zero length if it exists (TRUNCate)
nocreate	Error when opening if file does not already exist
noreplace	Error when opening for output if file already exists, unless ate or app is set
binary	Open file in binary (not text) mode

```
class person //class of persons
{
    protected:
        char name[80]; //person's name
        int age; //person's age
    public:
        void getData() //get person's data
        {
           cout << "\n Enter name: "; cin >> name;
            cout << " Enter age: "; cin >> age;
        }
        void showData() //display person's data
        {
            cout << "\n Name: " << name;
            cout << "\n Age: " << age;
        }
};
```

```
int main()
{
    char ch;
    person pers; //create person object
    fstream file; //create input/output file
    file.open("GROUP.DAT",  ios::app  |  ios::out  |  ios::in  |
          ios::binary );
    do //data from user to file
    {
        cout << "\nEnter person's data:";
        pers.getData(); //get one person's data
        file.write( reinterpret_cast<char*>(&pers), sizeof(pers) );
        cout << "Enter another person (y/n)? ";
        cin >> ch;
    } while(ch=='y'); //quit on 'n'
```

```
    file.seekg(0); //reset to start of file
    file.read( reinterpret_cast<char*>(&pers), sizeof(pers) );
    while( !file.eof() ) //quit on EOF
    {
        cout << "\nPerson:"; //display person
        pers.showData(); //read another person
        file.read(                  reinterpret_cast<char*>(&pers),
          sizeof(pers) );
    }
    cout << endl;
    return 0;
}
```

File Pointers

- Each file object has associated with it two integer values called the get pointer and the put pointer. These are also called the current get position and the current put position, or—if it's clear which one is meant—simply the current position.
- The seekg() and tellg() functions allow you to set and examine the get pointer, and the seekp() and tellp() functions perform these same actions on the put pointer.

```
class person //class of persons
{
    protected:
       char name[80]; //person's name
       int age; //person's age
    public:
        void getData() //get person's data
        {
            cout << "\n Enter name: "; cin >> name;
            cout << " Enter age: "; cin >> age;
        }
        void showData(void) //display person's data
        {
            cout << "\n Name: " << name;
            cout << "\n Age: " << age;
        }
};
```

```
int main()
{
    ifstream infile; //create input file
    infile.open("GROUP.DAT", ios::in | ios::binary);
    infile.seekg(0, ios::end); //go to 0 bytes from end
    int endposition = infile.tellg(); //find where we are
    int n = endposition / sizeof(person);
    cout << "\nThere are " << n << " persons in file";
    return 0;
}
```

Error Handling in File I/O

- In the file-related examples so far we have not concerned ourselves with error situations. In particular, we have assumed that the files we opened for reading already existed, and that those opened for writing could be created or appended to. We've also assumed that there were no failures during reading or writing.

```
const int MAX = 1000;
int buff[MAX];
int main()
{
    for(int j=0; j<MAX; j++) //fill buffer with data
        buff[j] = j;
    ofstream os; //create output stream
    os.open("a:edata.dat", ios::trunc | ios::binary);
    if(!os)
        { cerr << "Could not open output file\n"; exit(1); }
    cout << "Writing...\n"; //write buffer to it
    os.write( reinterpret_cast<char*>(buff), MAX*sizeof(int) );
    if(!os)
        { cerr << "Could not write to file\n"; exit(1); }
    os.close(); //must close it
```

```
ifstream is; //create input stream
is.open("a:edata.dat", ios::binary);
if(!is)
    { cerr << "Could not open input file\n"; exit(1); }
cout << "Reading...\n"; //read file
is.read( reinterpret_cast<char*>(buff), MAX*sizeof(int) );
if(!is)
    { cerr << "Could not read from file\n"; exit(1); }
for(j=0; j<MAX; j++) //check data
    if( buff[j] != j )
{ cerr << "\nData is incorrect\n"; exit(1); }
cout << "Data is correct\n";
return 0;
}
```

File I/O with Member Functions

- So far we've let the main() function handle the details of file I/O. When you use more sophisticated classes it's natural to include file I/O operations as member functions of the class.

```
class person //class of persons
{
    protected:
        char name[40]; //person's name
    int age; //person's age
    public:
        void getData(void) //get person's data
        {
            cout << "\n Enter last name: "; cin >> name;
            cout << " Enter age: "; cin >> age;
        }
        void showData(void) //display person's data
        {
            cout << "\n Name: " << name;
            cout << "\n Age: " << age;
        }
```

```
    void diskIn(int); //read from file
    void diskOut(); //write to file
    static int diskCount(); //return number of persons in file
};
void person::diskIn(int pn) //read person number pn
{ //from file
    ifstream infile; //make stream
    infile.open("PERSFILE.DAT", ios::binary); //open it
    infile.seekg( pn*sizeof(person) ); //move file ptr
    infile.read( (char*)this, sizeof(*this) ); //read one person
}
```

```
void person::diskOut() //write person to end of file
{
    ofstream outfile; //make stream
    outfile.open("PERSFILE.DAT", ios::app | ios::binary);
    outfile.write( (char*)this, sizeof(*this) ); //write to it
}
int person::diskCount() //return number of persons
{ //in file
    ifstream infile;
    infile.open("PERSFILE.DAT", ios::binary);
    infile.seekg(0, ios::end); //go to 0 bytes from end
    return (int)infile.tellg() / sizeof(person);
}
```

```
int main()
{
    person p; //make an empty person
    char ch;
    do { //save persons to disk
        cout << “Enter data for person:”;
        p.getData(); //get data
        p.diskOut(); //write to disk
       cout << “Do another (y/n)? “;
       cin >> ch;
   } while(ch==’y’); //until user enters ‘n’
   int n = person::diskCount();
```

```
    cout << “There are “ << n << “ persons in file\n”;
    for(int j=0; j<n; j++) //for each one,
    {
        cout << “\nPerson “ << j;
        p.diskIn(j); //read person from disk
        p.showData(); //display person
     }
    cout << endl;
    return 0;
}
```

Overloading the Extraction and Insertion Operators

- Let's move on to another stream-related topic: overloading the extraction and insertion operators. This is a powerful feature of C++. It lets you treat I/O for user-defined data types in the same way as basic types.
- We can overload the extraction and insertion operators so they work with the display and keyboard (cout and cin) alone. With a little more care, we can also overload them so they work with disk files.

```
class Distance //English Distance class
{
   private:
      int feet;
      float inches;
   public:
      Distance() : feet(0), inches(0.0)
      { }
     Distance(int ft, float in) : feet(ft), inches(in)
      { }
      friend istream& operator >> (istream& s, Distance& d);
      friend ostream& operator << (ostream& s, Distance& d);
};
```

```
istream& operator >> (istream& s, Distance& d)
{
    char dummy;
    s >> d.feet >> dummy >> dummy >> d.inches >>
        dummy;
    return s; //overloaded
}
ostream& operator << (ostream& s, Distance& d)
{
    s << d.feet << "\'-" << d.inches << '\"';
    return s;
}
```

```
int main()
{
    char ch;
    Distance dist1;
    ofstream ofile; //create and open
    ofile.open("DIST.DAT"); //output stream
    do {
        cout << "\nEnter Distance: ";
        cin >> dist1; //get distance from user
       ofile << dist1; //write it to output str
       cout << "Do another (y/n)? ";
       cin >> ch;
    } while(ch != 'n');
    ofile.close(); //close output stream
```

```
    ifstream ifile; //create and open
    ifile.open("DIST.DAT"); //input stream
    cout << "\nContents of disk file is:\n";
    while(true)
    {
        ifile >> dist1; //read dist from stream
        if( ifile.eof() ) //quit on EOF
        break;
        cout << "Distance = " << dist1 <<endl;
    }
    return 0;
}
```