# iostream

- istream, ostream classes
- cin, cout, cerr: object instances
- manipulators, state bits
- files: fstream
- random access
- istrstream, ostrstream

# streams

- We've already been doing "casual" i/o. Let's now see how it works, and how we can do more powerful things.
- In C++, all i/o uses the concept of a **stream**. A stream is simply a flow of bytes to/from a console, file, etc.
- A stream is an object – it can be constructed, inherited, etc.
- To help us, when `main` is called, the constructors for the iostream objects: `cin`, `cout`, `cerr` have *already* been called.
- *All* iostream objects need the `iostream.h` header file (and sometimes others.

# cerr

Very often, we send the output of one program into a file, or another program with redirection or pipes.
It is then useful to separate "normal" output from "abnormal" output.

This is done with the pre-defined output stream object `cerr`.

- the `cout` stream can be redirected
- the `cerr` stream is never redirected

```
#include <iostream.h>   // cerr.cc

int main() {
  cout << "This line can be redirected" << endl;
  cerr << "This one cannot." << endl;
  return 0;
}
```

---

First run this program in the normal way, then run it redirecting its output to a file:

```
% ./cerr
% ./cerr > foo.tmp
```

# streams as objects

- Since `cout` etc. are objects, they can (and do) have member functions. Take a look at `iostream.h` to see some of them.
- For instance, sometimes we want to write a "new line" character in a `for` loop, then flush the buffer once we're done. The new line character is `\n`.
- We use the `flush()` member function.
- If we are writing to a file, flushing can be expensive, so this is not so silly.
- We will see more member functions later.

```
#include <iostream.h>   // cout.flush.cc

int main() {
  for (int i=0; i<10; i++) {
    cout << "i, i^2, i^3: " << i << ", " << i*i
 << ", " << i*i*i << "\n";
  }
  cout.flush();
  return 0;
}
```

# Other "escape" characters

For completeness, here are other "escape" characters that can be used for beautifying your output.

| char | meaning | char | meaning |
|---|---|---|---|
| `\a` | bell | `\\` | backslash |
| `\b` | backspace | `\?` | question mark |
| `\f` | formfeed | `\'` | single quote |
| `\n` | newline | `\"` | double quote |
| `\r` | carriage return | `\oOO` | octal number OO |
| `\t` | horizontal tab | `\xHH` | hex number HH |
| `\v` | vertical tab | | |

# Manipulators and i/o Flags

- Eventually, we need to do more fancy footwork formatting the output. C++ allows us to do this with stream **manipulators**.
- The only hard part is there is more than one way to do it.

  A stream member function returns a stream – so we can simply put the function in our `cout` statement.

We distinguish 2 kinds of operation:

1. Operations that change the *state* of a stream.
   This is done with `setiosflags` and reset (to the default) with `resetiosflags`.

2. Operations that change the *next* item. They are:

| manipulator | purpose |
|---|---|
| setbase | sets the base |
| setfill | sets the fill character |
| setprecision | sets the precision |
| setw | sets the field width |

These manipulators affect the *next* item only. They can get confusing. The best thing is to "suck it and see".

# (re)setiosflags

The gory details can be found by poking through `iomanip.h`, `iostream.h`, and maybe `libio.h`, and `streambuf.h`.

Some of the flags (which can be OR'd together) are:

| flag | purpose |
|---|---|
| skipws | skip white space |
| left | left justify |
| right | right justify |
| dec | decimal |
| oct | octal |
| hex | hexadecimal |
| showbase | show the base |
| showpoint | show decimal point |
| uppercase | write uppercase |
| showpos | show positive sign |
| scientific | scientific notation |
| fixed | fixed notation |

It can get mighty confusing. To see how they work, let's "Just Do It".

---

```
#include <iostream.h>   // setiosflags.cc
#include <iomanip.h>

int main() {
  const int i=14;
  const double c=2.99792458e8;
  cout << c << " " << i << endl;
  cout << setiosflags(ios::uppercase) << "foo "
       << hex << c << " " << i << endl;
  cout << setiosflags(ios::fixed) << resetiosflags(ios::uppercase)
       << c << " " << i << endl;
  cout << setiosflags(ios::scientific | ios:: showpos) << oct
       << c << " " << i << endl;
  return 0;
}
```

# manipulators

```
#include <iostream.h>   // manipulators.cc
#include <iomanip.h>
int main() {
  const int i=14;
  const double c=2.99792458e8;
  cout << c << " " << i << endl;
  cout << setiosflags(ios::scientific)<<setw(20)<<setprecision(3)
       << c << " " << i << endl;
  cout << c << " " << i << endl;
  cout << setw(20) << setprecision(10) << c << " " << i << endl;
  cout << setiosflags(ios::fixed | ios::showpoint)
       << setw(20) << setprecision(12) << c << " " << i << endl;
  cout << setfill('#') << c << " " << setw(6) << i << endl;
  cout << setiosflags(ios::left) << c << " "<<setw(6)<<i<< endl;
  return 0;
}
```

# state bits

After an i/o operation, the stream objects set state bits, that can be used to examine the state of the last operation.

The *easiest* way is simply to test the stream itself:

```
#include <iostream.h>   // state.cc
#include <iomanip.h>

int main() {
  int i;
  cout << setiosflags(ios::uppercase);
  while (cin>>i)
    cout << dec << i << " = 0x" << hex << i << endl;
  return 0;
}
```

This works, because when EOF (`^D`) is reached, `cin` returns "false".

We can also read the state bits:

---

```
#include <iostream.h>   // rdstate.cc
#include <iomanip.h>

int main() {
  int io(cout.rdstate() & ios::badbit);
  cout << "io state = " << io << endl;
  return 0;
}
```

testing them against:

| |
|---|
| `ios::goodbit` |
| `ios::eofbit` |
| `ios::failbit` |
| `ios::badbit` |

# fstream

Output so far has used the `istream` and `ostream` objects. The base class for these is `ios`.
`ios` is also the base class for the file stream objects, `istream` and `ostream`, defined in `fstream.h`.

To see how it works, let's do an example:

---

```
#include <fstream.h>   // ofstream1.cc

int main() {
  ofstream out("foo.tmp");   // ofstream object "out"
  out << "Hello World" << endl;
  out.close();
  return 0;
}
```

Once we've called the `ofstream` constructor, we use the object *exactly* as we would any other stream.

The file is automatically closed when the object goes out of scope, but it doesn't hurt to see the close method.

We could use the constructor with other arguments:

---

```
#include <fstream.h>   // ofstream2.cc
#include <stdlib.h>
#include <time.h>

int main() {
  srand(time(0));
  ofstream out("foo.tmp", ios::app, 0644); // append
  out << "A random number: " << rand() << endl;
  out.close();
  return 0;
}
```

Or sometimes it's convenient to modify the object *after* calling the constructor:

---

```
#include <fstream.h>   // ofstream3.cc

int main() {
  ofstream out;
  out.open("foo.tmp");
  out << "Yowzer!" << endl;
  out.close();
  return 0;
}
```

We can equally easily do the same for input stream objects, using `ifstream`:

```
#include <fstream.h>   // ifstream1.cc
#include <String.h>

int main() {
  String s;
  ifstream in("poem.txt", ios::in);
  while ( in >> s )
    cout << s << endl;
  return 0;
}
```

Note how the string (we have surreptitiously introduced the `String` class) became "tokenized", which is probably not what we wanted.

There are *zillions* of ways of reading in a file, depending what we want to do.

---

```
#include <fstream.h>   // ifstream2.cc
#include <String.h>

int main() {
  String s;
  ifstream in("poem.txt", ios::in);
  while ( in >> s )
    cout << s << endl;
  return 0;
}
```

or we can read the file character by character:

---

```
#include <fstream.h>   // ifstream3.cc

int main() {
  char c;
  ifstream in("poem.txt", ios::in);
  while ( in.get(c) )
    cout << c;
  return 0;
  cout.flush();
}
```

If it's a file of numbers, we might want to read in the formatted numbers:

---

```
#include <fstream.h>   // ifstream4.cc

int main() {
  int i;
  ifstream in("numbers.txt", ios::in);
  while ( in>>i )
    cout << i << " ";
  cout.flush();
  return 0;
}
```

Note that altho the input file has spaces and <CR>, these get swallowed up.

Unless we explicitly test for a character, all "white space" is equivalent. White Space is any combination of:

- blank
- tab
- `\n`
- `\0`

Sometimes, we might want first to read the numbers into a string (or array of `char`), and then pick the string apart, character by character.

# random access

- The `fstream` classes also give us nice ways of doing random access by manipulating file pointers.
- We use `seekp(p)` to move the stream position to `p` for a put, and `seekg(p)` to move the stream position to `p` for a get.

```
#include <fstream.h>   // ifstream5.cc
#include <String.h>

int main() {
  char c;
  ifstream in("poem.txt", ios::in);
  streampos p;
  while (1) {
    cout << "Enter an offset: " << ends;
    cin >> p;
    in.seekg(p);
    in.get(c);
    cout << "The " << p << "'th byte is: " << c << endl;
  }
  return 0;
}
```

# istrstream, ostrstream

- The final i/o group contains the `istrstream, ostrstream` classes.
- These are "string streams" – we can do i/o on contents of memory in exactly the same way as on files.
- Again, let's illustrate with an example:

```
#include <fstream.h>   // istrstream1.cc
#include <strstream.h>
#include <String.h>

int main() {
  String s;
  char c;
  ifstream in("poem.txt", ios::in);
  while ( readline(in,s) ) {
    istrstream str( s.chars(), s.length());
    while ( str.get(c) )
      cout << c;
    cout << endl;
  }
  return 0;
}
```

Of course, we could also have done this by manipulating an array of `char`, but that is not so OO.

We can even do something useful: suppose we want to read a file containing arbitrary integers and reals. How do we know whether to read an `int` or a `double`?

- Read each number as a `String` using `ifstream`
- Determine the type by searching for "."
- Then read the `String` using `istrstream` accordingly

Once we have met operator overloading and pointers, we can do even more powerful things with the `strstream` classes.

```
#include <fstream.h>   // istrstream2.cc
#include <strstream.h>
#include <String.h>
int main() {
  String s;
  int i;  double x;
  ifstream in("mixed-numbers.txt", ios::in);
  while ( in >> s ) {
    istrstream str( s.chars(), s.length());
    if ( s.contains('.') ) {
      str >> x;
      cout << s << " -> Float read: " << x << endl;  }
    else {
      str >> i;
      cout << s << " -> Integer read: " << i << endl; }
  }
  return 0;
}
```

# iostream genealogy