Chapter 12. Miscellaneous classes

Authors: *Joseph T. Buck*

Other Contributors: *Yuhong Xiong*

This section includes classes that did not fit elsewhere.

12.1 Mathematical classes

12.1.1 Class Complex

Class Complex is a simple subset of functions provided in the Gnu and AT&T complex classes. The standard arithmetic operators are implemented, as are the assignment arithmetic operators `+=, -=, *=,` and `/=,` and equality and inequality operators == and `!=.` There is also `real()` and `imag()` methods for accessing real and imaginary parts. It was originally written when libg++ was subject to the GPL. The current licensing for libg++ does not prevent us from using it and still distributing Ptolemy the way we want, but having it makes ports to other compilers (e.g. cfront) easier. The following non-member functions take Complex arguments:

`Complex conj(const Complex&` *`arg`*`);`
`double real(const Complex&` *`arg`*`);`
`double imag(const Complex&` *`arg`*`);`
`double abs(const Complex&` *`arg`*`);`
Return the conjugate, real part, imaginary part, or absolute value, respectively.

`double arg(const Complex&` *`arg`*`);`
Return the angle between the X axis and the vector made by the argument. The expression
`abs(z)*exp(Complex(0.,1.)*arg(z))`
is in theory always equal to z.

`double norm(const Complex&` *`arg`*`);`
return the absolute value squared.

`Complex sin(const Complex&` *`arg`*`);`
`Complex cos(const Complex&` *`arg`*`);`
`Complex exp(const Complex&` *`arg`*`);`
`Complex log(const Complex&` *`arg`*`);`
`Complex sqrt(const Complex&` *`arg`*`);`
Standard mathematical functions. `log` returns the principal logarithm.

`Complex pow(double` *`base`*`,const Complex&` *`expon`*`);`
`xpon``);`
Raise base to expon power. There is also an operator to print a Complex on an ostream.

12.1.2 class Fraction

Class Fraction represents fractions. The header `Fraction.h` also provides declarations for the `lcm` (least common multiple) and `gcd` (greatest common divisor) functions, as these functions are needed for Fraction but are generally useful.

```
Fraction ();
Fraction (int num, int den=1);
```

The default constructor produces a fraction with numerator 0 and denominator 1. The other constructor allows the numerator and denominator to be specified arbitrarily.

```
int num() const;
int den() const;
```

Return the numerator or denominator.

```
operator double() const;
```

Return the value of the fraction as a double. Class Fraction implements the basic binary math operators `+, -, *, /;` the assignment operators `=, +=, -=, *=,` and `/=,` and the equality test operators `==` and `!=.` The method

```
Fraction& simplify();
```

reduces the fraction to lowest terms, and returns a reference to the fraction. There is also an operator to print a Fraction on an ostream.

12.2 Class IntervalList

The IntervalList class represents a set of unsigned integers, represented as a series of intervals of integers that belong to the set. It is built on top of a class Interval that represents a single interval. There is also a text representation for IntervalLists. This representation can be used to read or write IntervalList objects to streams, and also can be used in the IntervalList constructor. This text representation looks exactly like the format the “rn” newsreader uses to record which articles have been read in a Usenet newsgroup (which is where we got it from; thank you, Larry Wall). In the text representation, an IntervalList is specified as one or more Intervals, separated by commas. An Interval is either an unsigned integer or two unsigned intervals with an intervening minus sign. Here is one possible IntervalList specification: 1-1003,1006,1008-1030,1050 White space is not permitted in this representation. IntervalList specifiers do not have to be in increasing order, but if they are not, they are changed to "canonical form", in which any overlapping intervals are merged and the intervals are sorted to appear in increasing order. An IntervalList is best thought of as a set of unsigned integers. Its methods in many cases perform set operations: forming the union, intersection, or set difference of two IntervalLists.

12.2.1 class Interval and methods

The Interval class is in some ways simply an implementation detail of class IntervalList, but since its existence is exposed by public methods, it is documented here. An Interval has an *origin* and a *length,* and represents the set of *length* unsigned integers beginning with *origin.* It also has a pointer that can point to another Interval. The constructor

`Interval(unsigned` *origin*`=0, unsigned` *length*`=0,`

```
    Interval* nxt = 0);
```
permits all these members to be set. The copy constructor copies the origin and length values but always sets the next pointer to null. A third constructor

```
Interval(const Interval& i1,Interval* nxt);
```
permits a combination of a copy and a next-pointer initialization. The members

```
unsigned origin() const;
unsigned length() const;
```
return the origin and length values.

```
unsigned end() const;
```
The `end` function returns the last unsigned integer that is a member of the Interval; 0 is returned for empty Intervals. There are a number of queries that are valuable for building a set class out of Intervals:

```
int isAfter(const Interval &i1) const;
```
`isAfter` returns true if this Interval begins after the end of interval *i1* .

```
int endsBefore(const Interval &i1) const;
```
`endsBefore` returns true if this Interval ends strictly before the origin of interval i1.

```
int completelyBefore(const Interval &i1) const;
```
`completelyBefore` returns true if `endsBefore` is true and there is space between the intervals (they cannot be merged).

```
int mergeableWith(const Interval& i1) const;
```
`mergeableWith` returns true if two intervals overlap or are adjacent, so that their union is also an interval.

```
int intersects(const Interval& i1) const;
```
`intersects` returns true if two intervals have a non-null intersection.

```
int subsetOf(const Interval& i1) const;
```
`subsetOf` returns true if the argument is a subset of this interval.

```
void merge(const Interval& i1);
```
`merge` alters the interval to the result of merging it with *i1.* It is assumed that `mergeableWith` is true.

```
Interval& operator&=(const Interval& i1);
```
This Interval is changed to the intersection of itself and of *i1.*

12.2.2 IntervalList public members

```
IntervalList();
```
The default constructor produces the empty IntervalList.

```
IntervalList(unsigned origin,unsigned length);
```

This constructor creates an IntervalList containing *length* integers starting with *origin.*

`IntervalList(const char* definition);`
This constructor takes a definition of the IntervalList from the string in *definition,* parses it, and creates the list of intervals accordingly. There is also a copy constructor, an assignment operator, and a destructor.

`int contains(const Interval& i1) const;`
The `contains` method returns 0 if no part of *i1* is in the IntervalList, 1 if *i1* is completely contained in the IntervalList, and -1 if *i1* is partially contained (has a non-null intersection).

`IntervalList& operator|=(const Interval& src);`
Add a new interval to the interval list.

`IntervalList& operator|=(const IntervalList& src);`
Sets the IntervalList to the union of itself and *src.*

`IntervalList operator&(const IntervalList& arg) const;`
The binary & operator returns the intersection of its arguments, which are not changed.

`IntervalList& subtract(const Interval& i1);`
`IntervalList& operator-=(const Interval& i1);`
Subtract the Interval *i1* from the list. That is, any intersection is removed from the set. Both the `subtract` and `-=` forms are equivalent.

`IntervalList& operator-=(const IntervalList &arg);`
This one subtracts the argument *arg* from the list (removes any intersection).

`int empty() const;`
Return `TRUE` (1) for an empty IntervalList, otherwise `FALSE` (0).

12.2.3 IntervalList iterator classes.

There are two iterator classes associated with IntervalList, IntervalListIter and CIntervalListIter. The only difference is that the latter iterator can be used with const IntervalList objects and returns pointers to const Interval objects; the former requires a non-const IntervalIList and returns pointers to Interval. These objects obey the standard iterator interface; the `next()` or `++` function returns a pointer to the next contained Interval; `reset` goes back to the beginning.

12.3 Classes for interacting with the system clock

These classes provide simple means of interacting with the operating system's clock – sleeping until a specified time, timing events, etc. They may be replaced with something more general. Class TimeVal represents a time interval to microsecond precision. There are two constructors:

```
TimeVal();
TimeVal(double seconds);
```

The first represents a time interval of zero. In the second case, the *seconds* argument is rounded to the nearest microsecond. These classes rely on features found in BSD-based Unix systems and newer System V Unix systems. Older System V systems tend not to provide the ability to sleep for a time specified more accurately than a second.

`operator double() const;`
This returns the interval value as a double.

```
TimeVal operator +(const TimeVal& arg) const;
TimeVal operator -(const TimeVal& arg) const;
TimeVal& operator +=(const TimeVal& arg);
TimeVal& operator -=(const TimeVal& arg);
```
These operators do simple addition and subtraction of TimeVals.

```
int operator >(const TimeVal& arg) const;
int operator <(const TimeVal& arg) const;
```
These operators do simple comparisons of TimeVals.

Class Clock provides a simple interface to the system clock for measurement of actual elapsed time. It has an internal TimeVal field that represents the starting time of a time interval.

`Clock();`
The constructor creates a Clock with starting time equal to the time at which the constructor is executed.

`void reset();`
This method resets the start time to “now”.

`TimeVal elapsedTime() const;`
This method returns the elapsed time since the last `reset` or the call to the constructor.

`int sleepUntil(const TimeVal& howLong) const;`
This method causes the process to sleep until *howLong* after the start time.