Brown University ENGN2912B Fall 2017

Scientific Computing in C++

Lecture 9 | Strings & Streams

The **string** class

The C-style string is somewhat problematic in that if one is going to pass around string variables, they have to be allocated on the heap. Only constant strings (**const char***) can be generated without concern for memory leaks. If the string is going to be altered during run time (**char***) then string memory management will be needed. The designers of C++ were aware of this difficulty and designed a built-in class, **string**, which can be created and passed around without concern. In order to use the string class it is necessary to include its header file as in the following example.

```
#include <string>
int main() {
  string a = "I am a", b = " and b";
  string ab = a + b;
  cout << ab <<
  endl; string ab +=
  b; cout << ab <<
  endl;
}
```

```
I am a and b
I am a and b and b
```

Note that a **string** variable type can be declared and initialized just like any other type. In this case **a** and **b** are initialized with the value of two constant strings, but **a** and **b** are not **const** and can be changed. What is going on here is that the string class has a constructor that takes an argument of type **const char***. Thus a constant string can be on the right hand side of the **=** sign. The string constructor will look like,

```
string(const char* str);
```

As shown in the example, the **string** class defines a set of concatenation operators. The operator forms are **+** and **+=**. There are also string comparison operators, as shown in the following examples.

```
string a = "a";
string b = "b";

bool a_eq_b   =  a == b;  // false
bool a_neq_b  =  a != b;  // true
```

Note that

```
string a = 'a';
```

Will fail because `'a'` is a **`char`** data type not **`const char*`.**

`string` provides methods to determine the size of the string as in the following example.

```
string s29 = "I have exactly 29 characters.";
unsigned n = s29.size(); // n==29
string null =  "";
n = null.size(); // n = 0;
bool is_null = null.empty();//  is_null == true
```

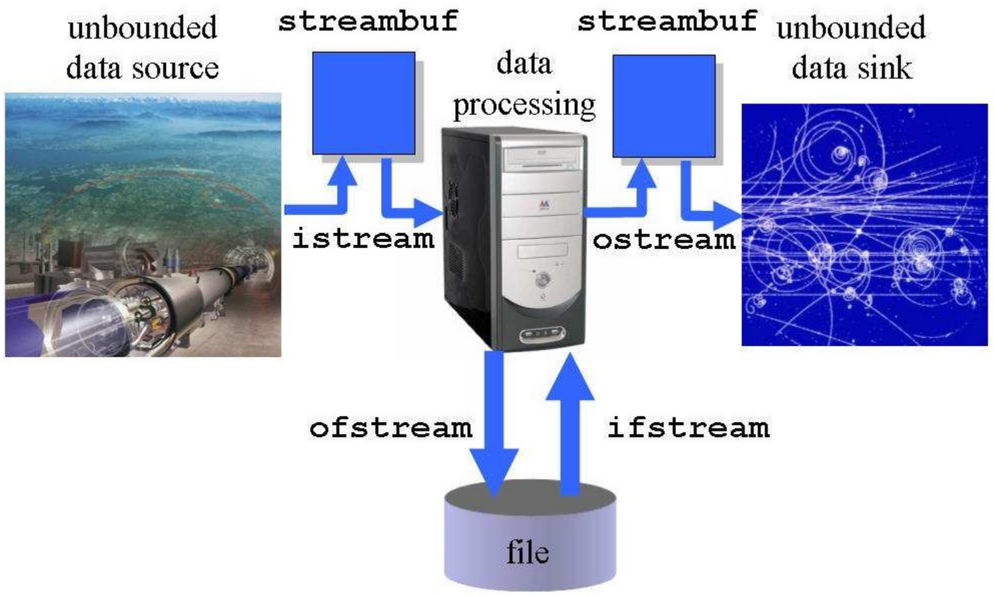
The method **`size`** reports how many actual characters there are in the string, including spaces. The method **`empty`** determines if the string has no characters**.**

The astute reader will wonder where the operations on the string class are that might be expected such as:

- finding a substring
- reversing the string
- inserting a string inside an existing string
- etc..

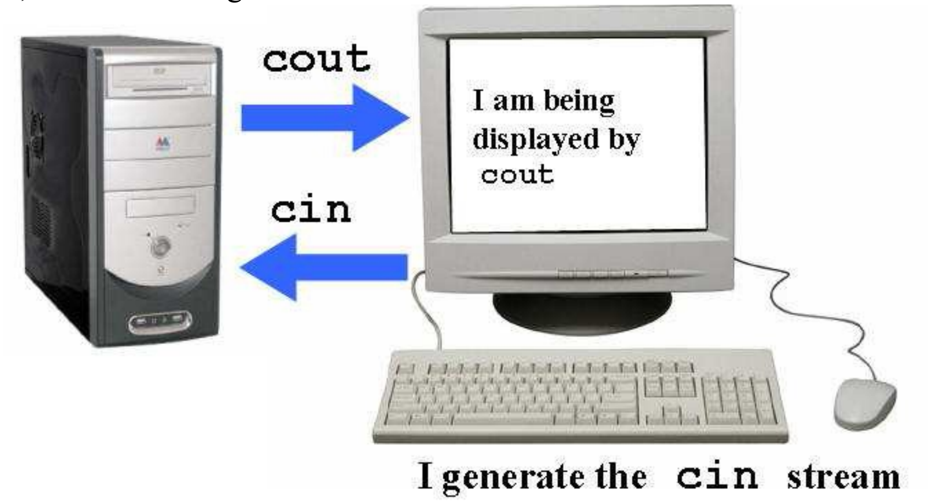
These methods, and many more, are actually available but they won't be understandable a this stage of development. They will require understanding the C++ standard template library (STL), which will be covered in a later lecture.

Streams



This figure illustrates the concept of a *stream*, which is a, possibly unbounded, linear sequence of data. The data could be continuously supplied by an instrument, processed to produce a result and then displayed. Another possibility is that the data is stored in a file and read, processed and the result stored back into the file. Streams often require buffering since there may be bursts and lags of data rate. The buffer smoothes out such rate fluctuations.

In C++, the stream is a built-in class which can supply or output a sequence of arbitrary data types, including classes of your own design. The simplest stream subclasses are **`cin`** and **`cout`**, shown in the figure below.



The streams cin and cout can represent a sequence of any data type but can also be interpreted as just a stream of type **`char`**, the primitive byte data type from which more structured data is composed. Perhaps a good way to start is to look at the primitive **`char`** input and output stream methods, using **`cin`** and **`cout`** as the example streams. Consider the following program.

```
#include <iostream>
char ch=0;
while( ch!='.') {
  cin.get(ch);
  cout.put(ch);
}
```

The input and output of this program is as follows.

```
type and press enter   <-keyboard echo as stream cin is created
type and press enter   <-result of cout
show the endpoint.
show the endpoint.
```

The first output is generated by the operating system that echoes the typed keystrokes. The input stream is captured in the buffer until the user types the “Enter,” or “Return” key, attaches the stream buffer to the program. The **`while`** loop then reads the stream until the **`'.'`** character is encountered on the stream.

There are several other useful primitive byte I/O operations for processing the stream.

```
char ch=0, skip = ',';
  while( ch!='.')
    if(cin.peek()== skip){
        cin.ignore()
      } else {
        cin.get(ch);
        cout.put(ch);
      }
```

The resulting input and output are as follows.

```
note that the ,,,s are gone. <-keyboard echo (the
input) note that the s are gone.  <-result of cout
```

The **peek** method on **cin** looks at the next **char** in the stream but doesn't actually remove it from the stream as is done by the **get** method. The other method, **ignore**, removes the next available **char** from the stream and discards it.

The primitive stream method, **getline**, is designed to acquire a given number of characters from the stream. Its use is illustrated below.

```
  char str[20];
  cin.getline(str, 20, '.');
  int nc= cin.gcount();
  for(int i = 0; i<=nc; ++i)
    cout.put(str[i]);
```

In the first input the number of characters was more than 20, which terminates **getline**.

```
Exceed twenty characters before a .
Exceed twenty charac
```

The general definition for **getline** is,
istream& getline (char* s, streamsize n, char delim)

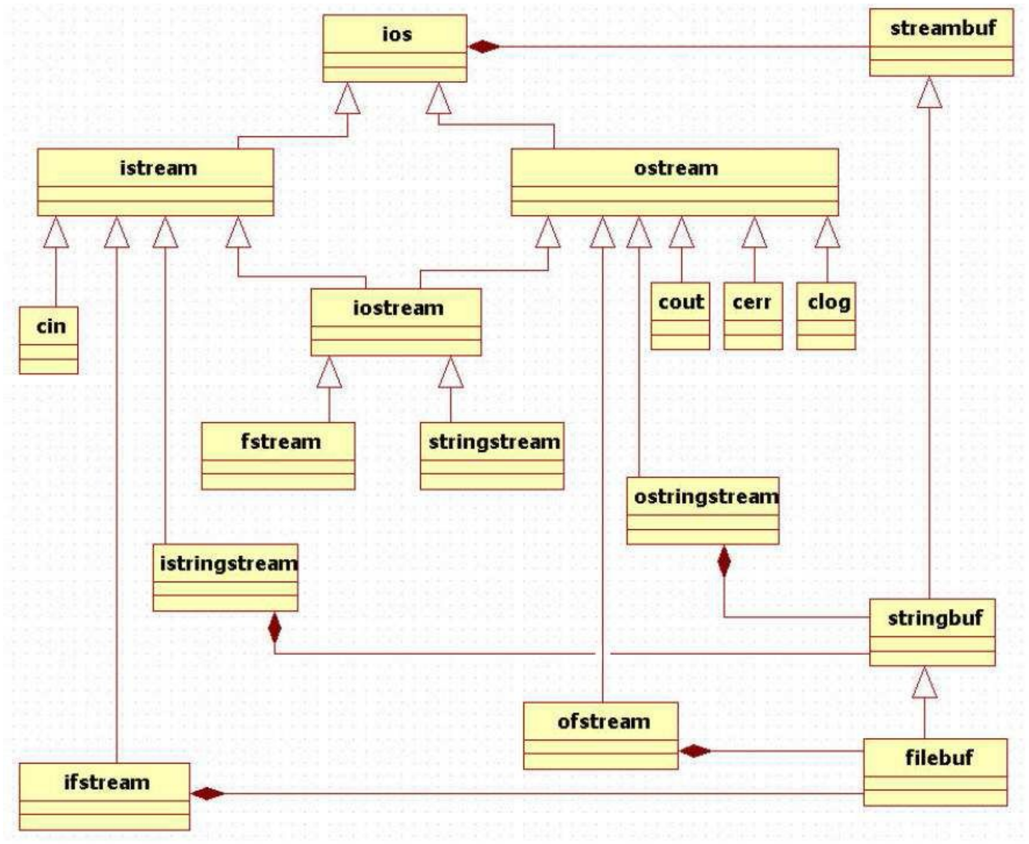
The third argument specifies a termination character for the line other than the default end of line character, **'\n'.** In this input/output **getline** is terminated instead by the **'.'**

```
Less than ten.
Less than ten
```

There are other primitive operations but most programs involving **char** (byte) streams can be implemented with the methods just presented. It is emphasized that all of these operations apply to any type of **iostream**, including file streams, which will be addressed later in this lecture.

The I/O stream class hierarchy

The following diagram shows the standard I/O stream class hierarchy.



There are obviously many capabilities that will only be partially covered in this course. The goal is to provide a basic understanding of streams so that routine programming tasks can be accomplished. The class hierarchy divides naturally between input and output streams, **`istream`** and **`ostream`**. There is another level of branching based on the embodiment of the stream. Two concrete embodiments are files and strings, leading to four subtypes, **`istringstream`**, **`ostringstream`**, **`ifstream`**, **`ofstream`**. Both are sequences of bytes and can represent the stream class.

It is also possible to have streams that one can input from and output to simultaneously. These stream classes multiply inherit from istream and ostream, to form **`stringstream`** and **`fstream`**.

The streams have associated buffers to hold data temporarily if the rate of production of data exceeds that of consumption. These buffers are naturally finite and so it is possible for a stream to loose data. This occurrence will trigger a termination of the program. The details of these buffer classes will not be covered in the course, since it is seldom necessary to know them. Note that **`cin`**, is a standard subclass of **`istream`** and **`cout`**, **`cerr`** and **`clog`** are standard subclasses of the **`ostream`** base class. The C++ language provides implementation for these subclasses, and reserved keywords that denote them in programs.

The input and output operators, >> and <<

The **`iostream`** library defines public two public operators, the stream input operator **>>** and the stream output operator **<<**. These operators interpret the stream in terms of types such as **`double`**, or **`point`**.

The output operator has already been used in many of the lecture examples. The output operator is extensively used to produce results from programs and will be described first. The signature of the operator is as follows.

```
ostream& operator<< ( ostream& os, type const& t );
```

The signature of the operator requires an **`ostream`** class reference (output stream) and a reference to the **`const`** variable to be output. Here ***`type`*** can be any defined type. Just to drive home the fact that the result of the operator is dependent on ***`type`***, consider the following example.

```
double dval = 3.14159;
unsigned uval = 314159;
cout << dval << ' ' << uval << endl;
```

```
3.14159 314159
```

The **<<** operator has interpreted three different data types, **`double`**, **`char`** and **`unsigned`** and displayed an appropriate view of their value. The **`double`** is printed with a decimal point, the **`char`** is interpreted as a space and the **`unsigned`** digits are presented as defined.

A close look at this example will reveal why the **<<** operator returns the ostream that it receives as an input. The statement **`cout << dval`** is interpreted by the compiler to assign **`cout`** as the **`ostream`** input, **`os`** and **`dval`** as the value of **`t`**. This relation can be made a clearer by placing the inputs to the stream above the operator symbol as below.

```
cout  dval
    <<    cout
```

The operator output, `cout`, is shown in light text below the input line. Since the operator **<<** returns its **`ostream`** input, the stream is passed along the chain of operators in the statement. This chaining is made more obvious below where The operator inputs are above the operator to the left and right.

```
cout  dval
    <<     cout    ' '
                <<      cout    uval
                             <<       cout     endl
                                          <<
```

endl, is a special stream operator called a *manipulator*. The signature for **endl** is as follows.

```
ostream&  endl( ostream& os);
```

The **endl** manipulator flushes any stream data from the internal stream buffer that hasn't yet been output and sends and end of line character, **'\n'** to the stream.

Other output stream manipulators

In general it is necessary to include the header file to access the full set of manipulators,
#include <iomanip>
The typical purpose of an **ostream** manipulator is to change how the stream displays or stores values.

The following table summarizes the set of **ostream** manipulators.

ostream manipulator name	**Action of the manipulator**
endl	Emit a new line and flush the ostream buffer.
setw(unsigned n)	Sets the minimum field width on output, a number with more digits will be longer than **n**.
width(unsigned n)	The same result as **setw.**
left	Left justifies the output, which is only meaningful if the width is set.
right	Right justifies the output, which is only meaningful if the width is set.
setfill(char c)	Fills unused spaces in the output, which is only meaningful if width is set.
setprecision(unsigned n)	For floating point, sets the precision (digits right of .)[1] of the output
fixed	Uses fixed point syntax. The precision defaults to 6 if not specified.
scientific	The alternative to **fixed**, uses power of 10 format.
boolalpha/noboolalpha	boolalpha outputs bool values as **"true"** and **"false,"** **noboolalpha** outputs **0, 1**
showpoint/noshowpoint	Shows the fractional part of a real number even if the fraction is zero.
uppercase/nouppercase	Specifies that hex letters and exponent **'e'** are output in upper case.
oct, dec, hex	Output an integer in base 8, base 10 or base 16 respectively
setbase(unsigned b)	Sets the base of the output to **b = 8 , 10, 16**
showbase/noshowbase	Indicates the base of the output (or not)
showpos/noshowpos	Output a **'+'** sign if the number is positive (or not)
internal	A number's sign is left justified and the number right justified.
flush	Forces the contents of the stream buffer to be output.
unitbuf/nounitbuf	Process when the buffer has any content vs. only process when full.
setiosflags(ios_base::fmtflags)/ resetiosflags(ios_base::fmtflags)	Control stream states by setting the flags directly.

The following program illustrates some of the effects of **ostream** manipulators.

[1] The total number of digits, or the number right of the '.' if the leading digit is 0.

```
#include <iostream>
#include <iomanip>

 const float tenth = 0.1;
 const float one   = 1.0;
 const float big   = 1234567890.0;
 const unsigned byte = 255;
 cout << "A. "                  << tenth << ", " << one << ", "
      << big << endl;
 cout << "B. " << fixed         << tenth << ", " << one << ", "
      << big << endl;
 cout << "C. " << scientific    << tenth << ", " << one << ", "
      << big << endl;
 cout << "D. " << fixed << setprecision(3) << tenth << ", "
      << one << ", " << big << endl;
 cout << "E. " << setprecision(20) << tenth << endl;
 cout << "F. " << setw(8) << setfill('*') << 34 << 45 << endl;
 cout << "G. " << 34 << setw(10) << 45 << endl;
 cout << "H. " << oct << byte << ", "<< dec << byte << ", "
       << hex << byte << endl;
 cout << "I. " << setprecision(3) << showpos << one << ", "
      << noshowpos << one << endl;
 cout << "J. " << byte << ", "<< resetiosflags(ios::hex)
      << byte << endl;
```

The output is as follows.

```
A. 0.1, 1, 1.23457e+009
B. 0.100000, 1.000000, 1234567936.000000
C. 1.000000e-001, 1.000000e+000, 1.234568e+009
D. 0.100, 1.000, 1234567936.000
E. 0.10000000149011612000
F. ******3445
G. 34********45
H. 377, 255, ff
I. +1.000, 1.000
J. ff, 255
```

The `istream` operator

The signature of the istream operator is,

`istream&  operator>>( istream& is,` ***`type`*** **`& t );`**

Note that this form is very similar to the ostream operator, except the type variable, **`t`**, is no longer a **`const`** reference. The type can't be const because **`t`** is now an output of the operator. Just as with the ostream operator the input byte stream is interpreted according to a format compatible with the specified ***`type`***. The following program provides a few examples of **`>>`**.

```
#include <iostream>
#include <iomanip>

 double plain,
 scientif; unsigned
 short decim, hexa;
 unsigned first,
 second;

 cin >>

 plain;
 cin >>

 scientif
 ;
 cin >> dec >>
 decim; cin >>
 hex >> hexa;
 cin >> dec >> first >> second;
```

The result of supplying inputs from the keyboard is as follows. The input to one application of the operator is terminated by the "enter" or "return" keystroke, or a white space.

```
Input                  Result
12.345          -> plain  == 12.345
1.2345e001      -> scientif
== 12.345 999   -> decim  ==
999
999             -> hexa == 0x0999 == 2457
101 102         -> first  == 101,second == 102
```

Note that the same chaining action of the **`>>`** operator output passes the stream to subsequent inputs. Note also that some stream manipulators such as **`hex`** apply to interpret the format of the input as well. The first input of **`999`** is interpreted as decimal **`999`**. The second input of **`999`** is interpreted as **`0x999`**.

There are several manipulators that apply only to **`istream`**.

`istream` **manipulator name**	**Action of the manipulator**
`skipws/noskipws`	Interpret white space e.g., space, tab as a data separator (or not)
`ws`	Reads a white space at the current stream location and discards it.

Implementing the stream operator for a new data type

So far the discussion has been centered around the built-in data types of C++. However the **>>** and **<<** operators can be implemented for any data type including classes. To illustrate how a stream operator is implemented, consider the class **`point`** from Lecture 6.

The following is the content of the modified `point.h` file.

```
#ifndef point_h_
#define point_h_
#include <iostream>
using namespace std;
class point {
 public:
   point();
   point(double x, double y);
   double x() const {return x_;}//implicitly inline
   double y() const {return y_;}
 private:
   friend ostream& operator<<(ostream&, point const&);
   friend istream& operator>>(istream&, point&);
   double x_;
   double y_;
};

inline ostream& operator<<(ostream& os, point const& p)
{
  os << p.x_ << ' ' << p.y_ << endl;
  return os;
}

inline istream& operator>>(istream& is, point& p)
{
  is >> p.x_ >>  p.y_;
  return is;
}
#endif  // point_h_
```

There are several new topics introduced by the implementation above. The first is the concept of a “friend.” As declared in the statement,

```
friend ostream& operator<<(istream&, point const&);
```

This declaration states that the operator **<<** can access the private members of the **`point`** class. Normally the point coordinates are to be kept private from direct access by other classes. Here the stream operators can access the coordinate values directly without expending a function call to get at them. This private access increases the throughput of stream I/O. In other words, the operator **<<** is a close *friend* of **`point`**, and is allowed private access.

The other new keyword is **`inline`**. This keyword is a suggestion to the compiler that the object code for the stream operators be inserted directly at the location the function is called rather than using a pointer to access the function. This inline compilation of the function code is reasonable if the function is short and requires minimum execution time. It should be noted that class member functions that are implemented within the class body are implicitly considered to be **`inline`**. For example the **`x()`** and **`y()`** accessors of point are implicitly **`inline`**.

Also note that short functions, e.g. the stream operators, which are not member functions can also be implemented directly in the `point.h` file. Normally this is discouraged to keep the compiler from processing the function implementations each time the .h file is included.

However there is a benefit to having the implementation details directly visible near the declaration, and for short functions, the overhead is negligible. Moreover, **`inline`** functions are not visible outside the file where they are implemented. By putting the implementation in the .h file, the implementation is always visible when needed.

An example use of the **`point`** stream operators is as follows.

```
#include <iostream>
#include point
point pa(1.0, 2.0), pb(3, 4), pc;
cout << pa << pb << endl;
cin >> pc;
```

The output of the program is

```
1.000 2.000
3.000 4.000
```

With an input of,

```
5 6
```

the result is that **`pc`** is constructed with **`x_==5`** and **`y_==6`**.

File I/O, `ofstream` and `ifstream`

The most common use of streams is to create and access data stored in files. The file input stream ifstream and output stream ofstream are defined by the include statement,

`#include <fstream>`

If only input is required, the following include is used

`#include <ifstream>`

and for output only,

`#include <ofstream>`

The following simple program illustrates the basics of file I/O.

```
#include <fstream>
ofstream ofstr("my_file");
if(ofstr){
    string s = "I am writing into my file";
    ofstr << s << endl;
    }
ofstr.close();
ifstream ifstr("my_file");
while(ifstr)
{
 string s;
 ifstr >> s;
 cout << s << endl;
}
```

If the specified file path exists, then the file is created and opened for writing. The value of **`ofstr`** will be **`true`** inside conditional statements. A file close, **`ofstr.close()`**, writes any existing buffer data to the file and **`ofstr`** has the value **`false`** in any subsequent conditional. The file **`"my_file"`** has the following contents after it is closed.

```
I am writing into my file
```

The output of the program is as follows.

```
I
am
writing
into
my
file
```

Note that the spaces in the string are interpreted as the end of the stream data item and so the **`while`** loop executes six times until the end of the file is reached. The file stream returns a value of **`false`** inside a conditional when the end of file is reached. This behavior makes reading the file easy, as with the **`while`** loop above.

It is possible to use a **`string`** to supply the filename as follows.

```
string file_path = "my_file";
ofstream ofstr(file_path.c_str());
```

It is necessary to use the method **`c_str()`** on **`string`** to return a **`const char*`** value as the argument of the **`ofstr`** constructor. Note that it is possible to open a file for both reading and writing by establishing a two way stream buffer. This capability requires additional specification using stream attributes defined as static members of the **`ios`** base class (see class hierarchy above). For example,

```
#include <fstream>
#include <iostream>
…

fstream fstr("file_name",ios::in|ios::out);//both in and out
bool op = fstr.is_open(); // op == true only if file_name exists
//write  a value
fstr << 100 <<
endl;
fstr.seekg(0);// move stream pointer to start of file
unsigned x;
fstr >> x;//read the value back
cout << "x is " << x << endl;//prints “x is 100”
fstr.close();// file now contains 100 on the first line
```

The `stringstream`

A stream can also be implemented by a *string* of characters and is called a **`stringstream`**. The input form of the **`stringstream`** is **`istringstream`** and output form, **`ostringstream`**. (see the class hierarchy above). The **`stringstream`** type can be used for both input and output, and has arbitrarily large capacity , which can expand to all of available memory. An example of creating a **`stringstream`** and using it as a output and input stream is shown below.

```
#include <sstr>
 double x = 1.3, y = 1.7, z = 1e-010;
 double u, v, w;
 stringstream str;
 str << x << ' ' << y << ' ' << z << ends;
 str >> u >> v >> w;
 string ss = str.str();
 cout << ss << endl;
```

The output of the program is shown below.

```
1.3 1.7 1e-010
```

Note that the **`stringstream`** output statement is terminated with **`ends`** and not **`endl`** as with files and **`cout`**. The string that used to implement the stream can be recovered by the method **`str()`**. This function is very useful for encoding a string with a mixture of data types.

As an example, consider the problem of generating a set of filenames which have a index as part of the name. This issue can arise for example in collecting a sequence of data readings where the file name indicates an index of the sequence.

```
string files[5];
for(unsigned i=0; i<5; ++i)
{
  stringstream strm;
  strm <<"base_file_name_" << i << ends;
  files[i]=strm.str();
  cout << files[i] << endl;
}
```

The output is as follows.

```
base_file_name_0
base_file_name_1
base_file_name_2
base_file_name_3
base_file_name_4
```

Note that the file name has a fixed base name and then an incrementing final segment. The array **`files`** can be used later in the program to open a series of files for writing.