Week 10 Lecture 2

NWEN 241
Systems Programming

Jyoti Sahni

jyoti.sahni@ecs.vuw.ac.nz

Content

- Vectors (from previous lecture)
- File Handling in C++
- Dynamic Memory Allocation

Recap: Generic Programming

- Generic programming involves writing code in a way that is **independent** of any particular type.

- It allows ***type*** as a parameter to methods and classes.

- A *type* parameter may be a primitive / built-in type such as `int` or `double` or a user defined type such as `class` or `structure.`

- Generics eliminates the need to create different algorithms if the data type is an integer, string or a character.

- Generics can be implemented in C++ using **Templates**. Templates allow us to create a single **function** or a **class** to work with different data types.

Recap: Standard Template Library

- Standard Template Library (STL) is a library for C++.

STL has with three basic components:

Containers

- Containers are used to manage collections of objects of a certain kind

Algorithms

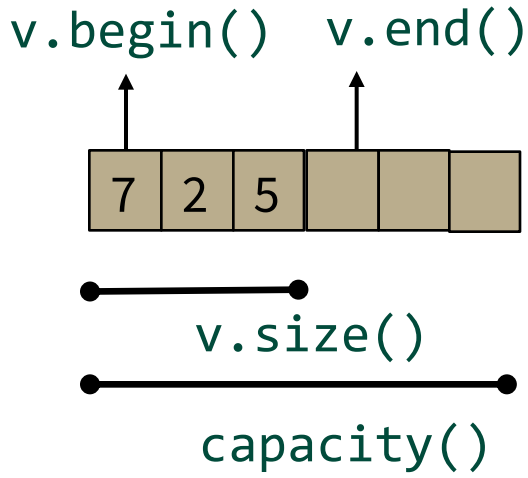
- Algorithms act on containers.
- Independent of containers

Iterators

- Generalized pointers that facilitate use of containers
- Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers

Recap: Vector

- One of the containers is **Vector.**
- Defined in `<vector>`
- Same as **dynamic arrays** with the ability to resize itself **automatically** when an element is inserted or deleted, with their storage being handled automatically by the container.



- Vector elements are placed in contiguous storage.
- The **capacity** of the vector is decided by the compiler. It is generally bigger than the **size** of elements in it.
- This gives the ability to quickly insert an element to the end or remove the last one, just by keeping track of the number of elements.
- Vectors also have safety features that make them easier to use than arrays, automated bounds checking and memory management

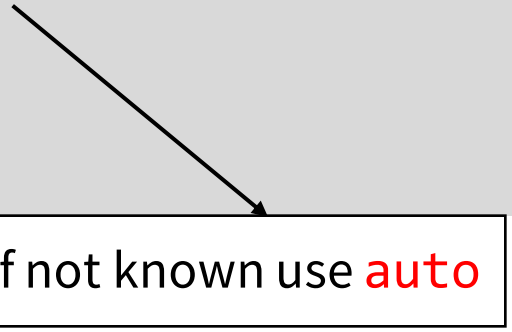
Vectors

Iterators	`begin()`	Returns an iterator pointing to the first element in the vector
	`end()`	Returns an iterator pointing to the theoretical element that follows the last element in the vector
Reverse Iterators	`rbegin()`	Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
	`rend()`	Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

https://en.cppreference.com/w/cpp/container/vector

Example

```
#include <iostream>
#include <vector>


int main(){
   vector <int> v ={1,2,3,4,5};

   cout<<"Incrementing the vector by 1:"<<endl;

   for (std::vector<int>::iterator it = v.begin() ; it != v.end(); ++it){
     *it = *it + 1;
      std::cout << ' ' << *it;
    }
}
```




```
Output:
Incrementing the vector by 1:
2 3 4 5 6
```

Basic Vector Operations

capacity	`empty()`	checks whether the container is empty
	`size()`	returns the number of elements
	`resize()`	resizes the container so that it contains *n* elements
	`capacity()`	returns the number of elements that can be held in currently allocated storage
	`shrink_to_fit()`	reduces memory usage by freeing unused memory

https://en.cppreference.com/w/cpp/container/vector

Basic Vector Operations

Element access	`at()`	access specified element with bounds checking (throws exception when a non-existent member is accessed)
	`operator[]`	access specified element (does not do range checking)
	`front()`	access the first element
	`back()`	access the last element

https://en.cppreference.com/w/cpp/container/vector

Basic Vector Operations

Modifiers	`assign()`	assigns new content
	`insert()`	inserts elements
	`erase()`	erases elements
	`clear()`	removes all elements from the vector
	`push_back()`	Adds a new element at the end of the vector, after its current last element
	`pop_back()`	removes the last element in the vector, effectively reducing the container size by one.
	`emplace()`	Adds a new element at a given position *inplace*

https://en.cppreference.com/w/cpp/container/vector

```
class A{
    int a;
    int b;

public:
    A(int x, int y){
         a=x;
         b=y;
}
    A(){}

void show(){
        cout<<"a = "<<a<<" "<<"b =
"<<b<<endl;
    }
```

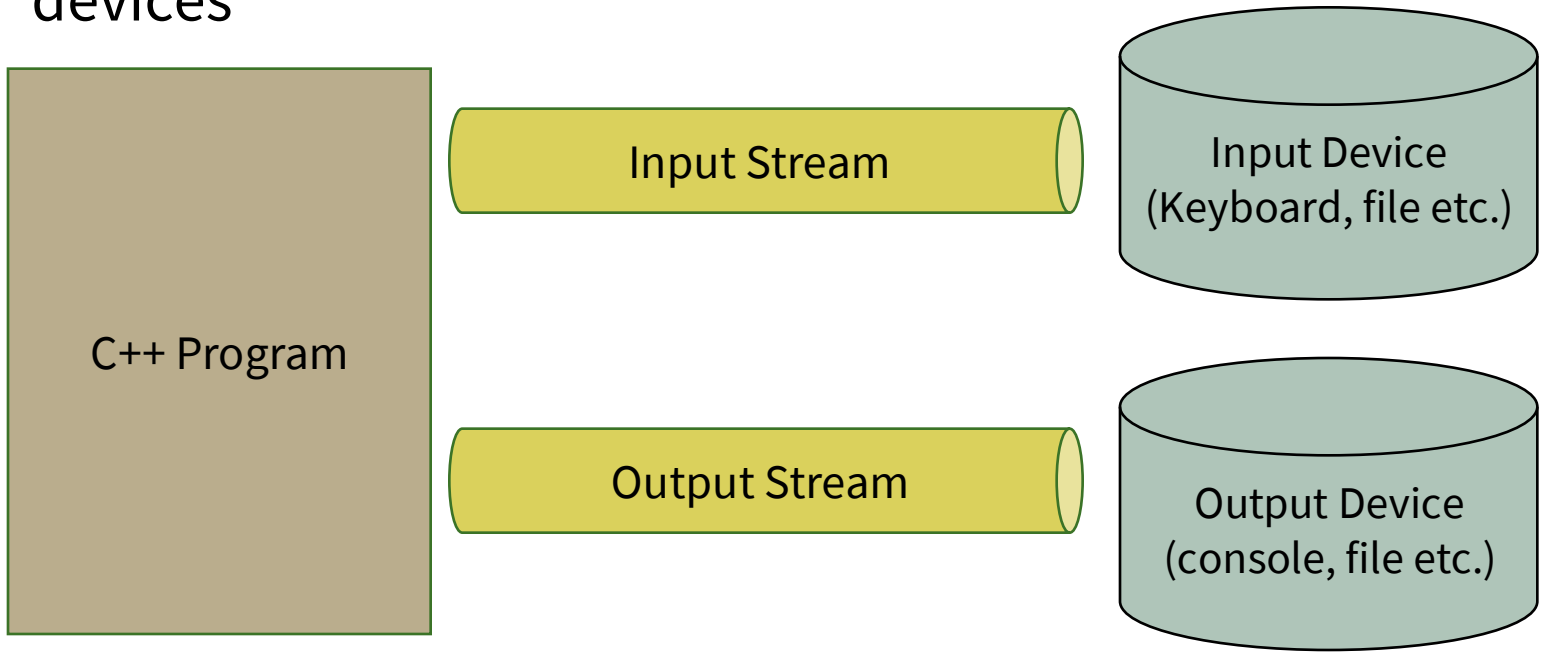
```
Size: 5
Capacity: 8
Element at Loc 0
a = 0 b = 0
Element at last Loc
a = 4 b = 4
Inserting a new element in the beginning
Element at Loc 0 is:
a = -1 b = -1
```

```
int main(){
vector<A> vecA;
 for (int i = 0; i <= 4; i++)
{
      vecA.push_back(A(i,i));
}
cout<<"Size: "<<vecA.size()<<endl;
cout<<"Capacity: "<<vecA.capacity()<<endl;

cout<<"Element at Loc 0"<<endl;
vecA[0].show();

cout<<"Element at last Loc"<<endl;
vecA.back().show();

cout<<"Inserting a new element in the beg"<<endl;
A a1(-1,-1);
vecA.insert(vecA.begin(),a1);

cout<<"Element at Loc 0 is: "<<endl;
vecA.at(0).show();
return 0;   }
```

File Handling in C++

Recap: I/O Basics

- C/C++ I / O are based on *streams*, which are sequence of bytes flowing in and out of the programs
- Streams acts as an intermediaries between the programs and the actual IO devices

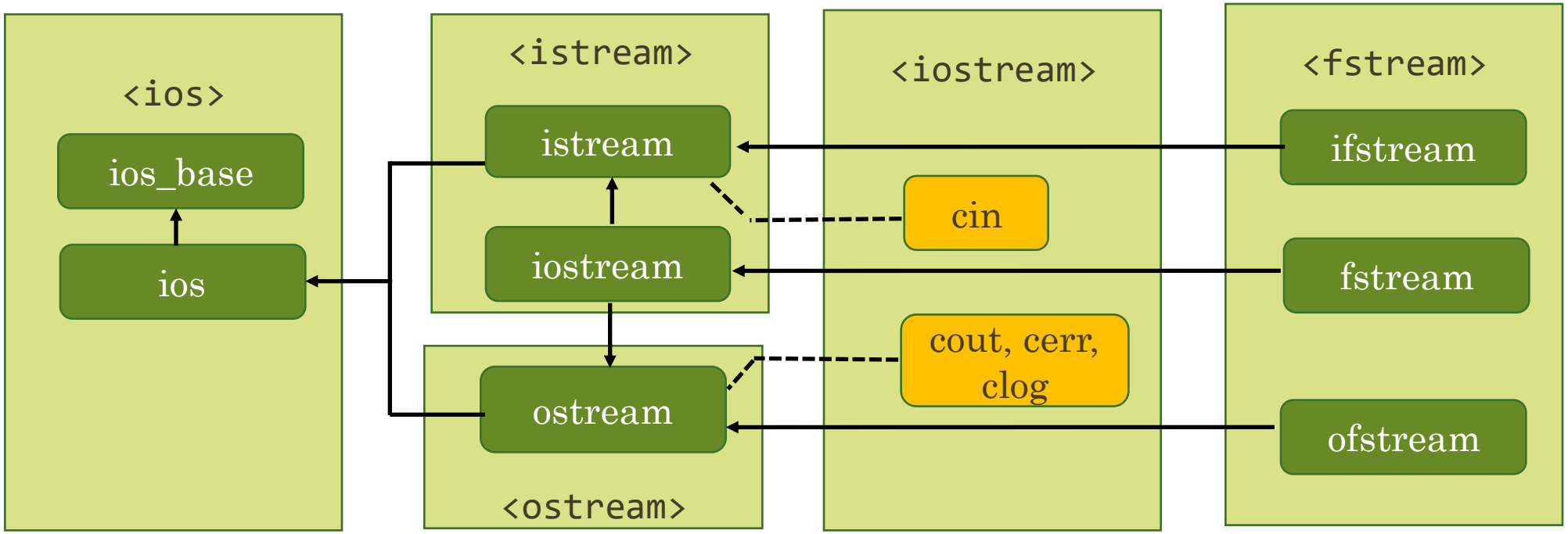


With **input** streams, the **extraction operator (>>)** is used to remove values from the stream

With **output** streams, the **insertion operator (<<)** is used to put values in the stream

C++ IO operations are *device independent*. The same set of operations can be applied to different types of IO devices.

Stream Hierarchy



A stream is represented by an object of a particular class.

File Streams

- `ifstream` – stream class to **read** from files
- `ofstream` – stream class to **write** to files.
- `fstream` - stream class to both **read (from) and write (to)** files.
- The stream objects `cin`, `cout`, `cerr` and `clog` are declared in `iostream` header file and are automatically added to our program, when `iostream` header file is included in our program.
- In contrast, we are responsible for creating and setting up our own file streams.

Steps for File IO:

1. Create file stream objects

```
ifstream  fsIn;  //input
ofstream fsOut;  // output
fstream fsBoth;  //input & output
```

2. Open the file

```
fsIn.open("data.txt",fileopenmode);
```

File open mode

OR Combine the two steps

```
ifstream fsIn("data.txt", fileopenmode);
```

File Open Modes

Name	Description
`ios::in`	Open file to read (default for ifstream)
`ios::out`	Open file to write (default for ofstream)
`ios::app`	Output operations happen at the end of the file, appending to its existing contents.
`ios::ate`	The stream's position indicator is set to the end of the file.
`ios::trunc`	Deletes all previous content in the file. (empties the file)
`ios::nocreate`	If the file does not exists, new file is not created
`ios::noreplace`	If the file exists, trying to open it with the open() function, returns an error.
`ios::binary`	Opens the file in binary mode.

`ios::in` is default for ifstream
`ios::out` is default for ofstream
`ios::in |ios::out` is the default for fstream

Examples

- Open a file named data.txt for reading

```
ifstream  fsIn;
fsIn.open("data.txt", ios::in);
```

- Open or **create** a file named mydata.csv for writing

```
ofstream  fsOut;
fsOut.open("data.txt", ios::out);
```

- To set more than one open mode, just use the **OR** operator- |.

```
fstream fsBoth
fsBoth.open("data.txt", ios::in | ios::out);
```

Steps for File IO:

3. Check if the file is opened properly. Every stream object has an `is_open( )` method that returns `"true"` if a file is open and associated with the stream object, otherwise it returns `"false"`.

```
if(fsOut.is_open())
    cerr<<"Failed to open file"
```

```
fsOut.bad() //true if reading writing operation fails
fsOut.fail() // true if bad + format errors
fsOut.eof() // true if file has reached the end
```

True if none of the stream's error state flags (eofbit, failbit and badbit) is set

```
if(fsOut.good())
    fsOut<<"Hello World";
```

4. Read and write to files using the functions defined in the stream's pubic interface

Steps for File IO:

5. Close the file after use.

```
fsIn.close(); //Close files
fsOut.close();
fsBoth.close();
```

Note:All `istreams` (and `ifstreams`) have a method **`eof( )`** that returns a boolean value of true if an attempt is made to read past the end of the file, and false otherwise.

Formatted Output

- Formatted output is carried out on streams using the stream insertion << operator for all basic and overloaded data types.

```
int a = 1 ;
char c = 'A';
string s =" Sam P";

ofstream fOut;
fOut.open("example.txt");

fOut << a << " " << c << " " <<s;
fOut.close();
```

example.txt

```
1 A  Sam P
```

Formatted Input

- Formatted input is carried out on streams using the stream extraction operator >> for all basic and overloaded data types.

```
int a1 ;
char c1 ;
string s1 ;

ifstream fIn;
fIn.open("example.txt");

fIn >> a1 >> c1 >> s1;
cout<< a1 << c1 << s1;
fIn.close();
```

example.txt

```
1 A Sam P
```

Output

```
1 A Sam
```

Note: >> (extraction operation) Operator does not consider white space characters as part of strings(white space characters causes extraction to terminate)

Unformatted IO

Single Character - Input	
Name	**Description**
`int get ();`	Extracts a character from a stream and returns as int.
`istream & get (char & c);`	Extracts a character from a stream, stores it in c and return the invoking istream reference

Single Character - Output	
Name	**Description**
`ostream & put (char c);`	Inserts character c into the stream and return the invoking ostream reference

Example

```
char c;
ofstream fOut;
fOut.open("Data.txt");

    while((c=cin.get())!='\n')
            fOut.put(c);

            fOut.close();
```

Input | Welcome NWEN241

Data.txt | Welcome NWEN241

int get() returns ASCII value of the character read

Example

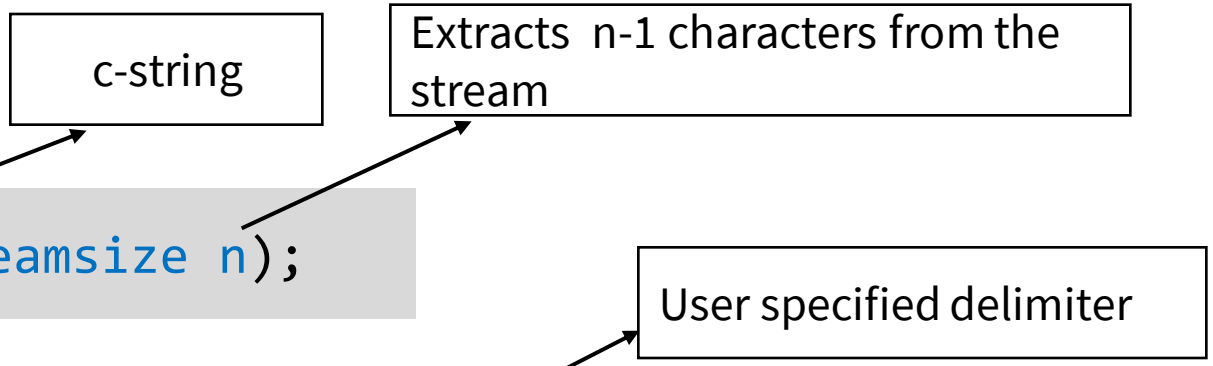
```
int main ()
{
  char cat, subcat;
  cout<<"Enter cat followed by subcat without space: ";

  cin.get(cat).get(subcat);

  cout<<"\nYou entered: ";
  cout.put(cat).put(' ').put(subcat);

  return 0;

 }
```

get(ch) and put(ch) can be cascaded, as they returns the invoking stream reference.

Unformatted IO

- C - String

1.
```
istream& get (char* s, streamsize n);
```



2.
```
istream& get (char* s, streamsize n, char delim);
```

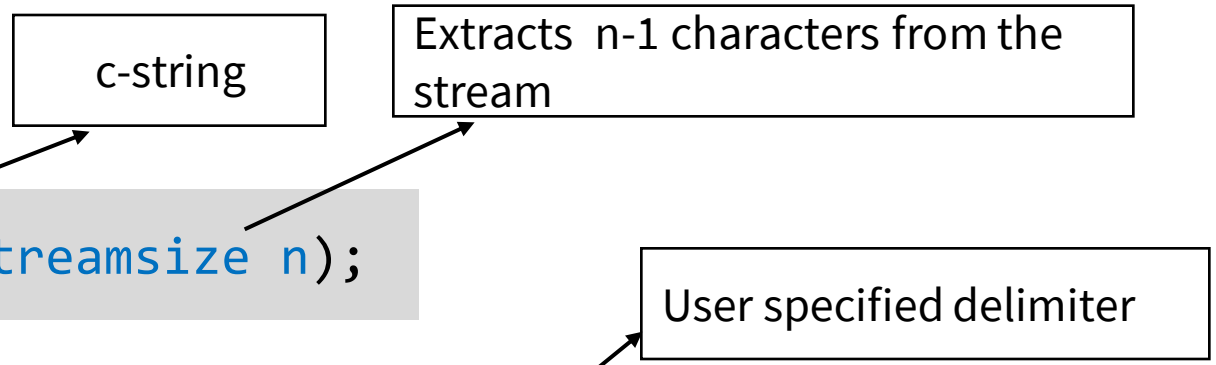
- `Default Delim` in (1) is `'\n'`
- Extracts characters until either the extracted character is `delim` or `n-1` characters have been read.
- **Appends** null character (`'\0'`) to s.
- **Keeps** the `delim` char in the input stream

Unformatted IO

- C - String

3.
```
istream& getline (char* s, streamsize n);
```

4.
```
istream& getline (char* s, streamsize n, char delim);
```



- Same as `get()`, but **extracts and discards** `delim` char from the input stream

Example

```
int main ()
{
    char c[20];
    cin.get(c,20);
    cout<<c;
 }
```

Input: Sam P

Output: Sam P

```
int main ()
{
   char c[20];
   cin.getline(c,20);
   cout<<c;
}
```

Input: Sam P

Output: Sam P

Example

```
int main ()
{
    char c[20];
    char c1;

    cin.get(c,20);
    cin.get(c1);

    cout<<c<<" "<<c1;
}
```

- `get(c, 20)` Keeps '\n' in the input stream.
- Reads `'\n'` into `c1`

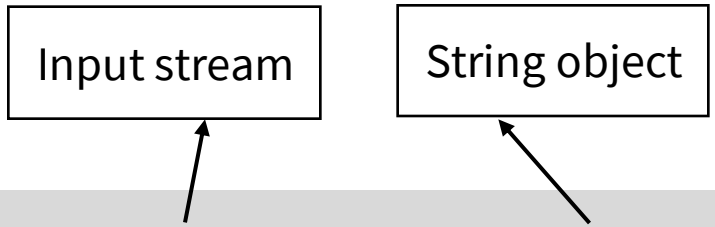
```
int main ()
{
    char c[20];
    char c1;

    cin.getline(c,20);
    cin.get(c1);

    cout<<c<<" "<<c1;
}
```

- `getline(c,20)` Discards '\n' from the input stream.

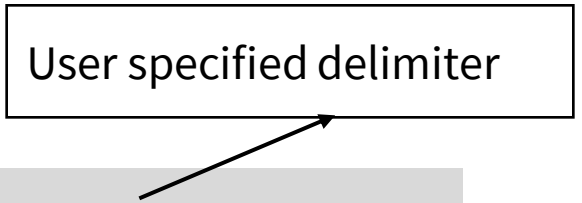
Unformatted IO

- String class

1.
```
istream& getline (istream& is, string& s);
```



2.
```
istream& getline (istream& is, string& s, char delim);
```



- `Default Delim` in (1) is `'\n'`
- Extracts characters until the extracted character is `delim` character is found.
- **Discards** the `delim` char

Example

```
int main ()
{
    string s;
    ofstream fOut;
    fOut.open("Data.txt");

     getline(cin,s);

     fOut<<s;
     fOut.close();
     return;
  }
```

Input

Welcome !!

Data.txt

Welcome !!

Random access

- Each file stream contains a **file stream pointer** to track current read / write position within a file.

- To know current position :
  - `tellg()` function (for input) and
  - `tellp()` function(for output) files

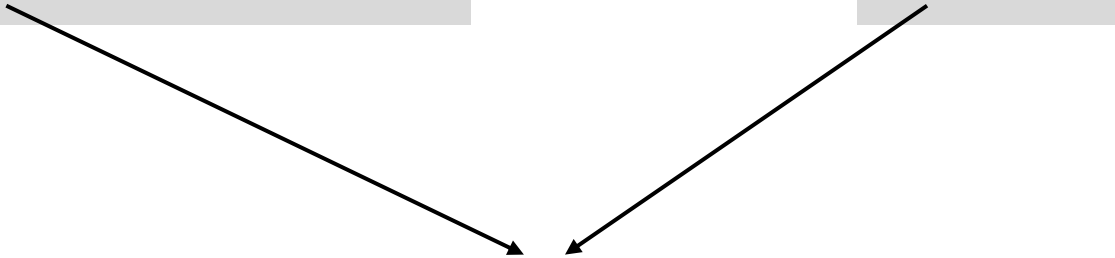
- For manipulating the file pointer:
  - `seekg()` for input and
  - `seekp()` for output.

Random access

- To know current position :
  - `tellg()` function returns the position of the current character in the input stream.
  - `tellp()` function returns the position of the current character in the output stream.

```
streampos tellg();
```

```
streampos tellp();
```



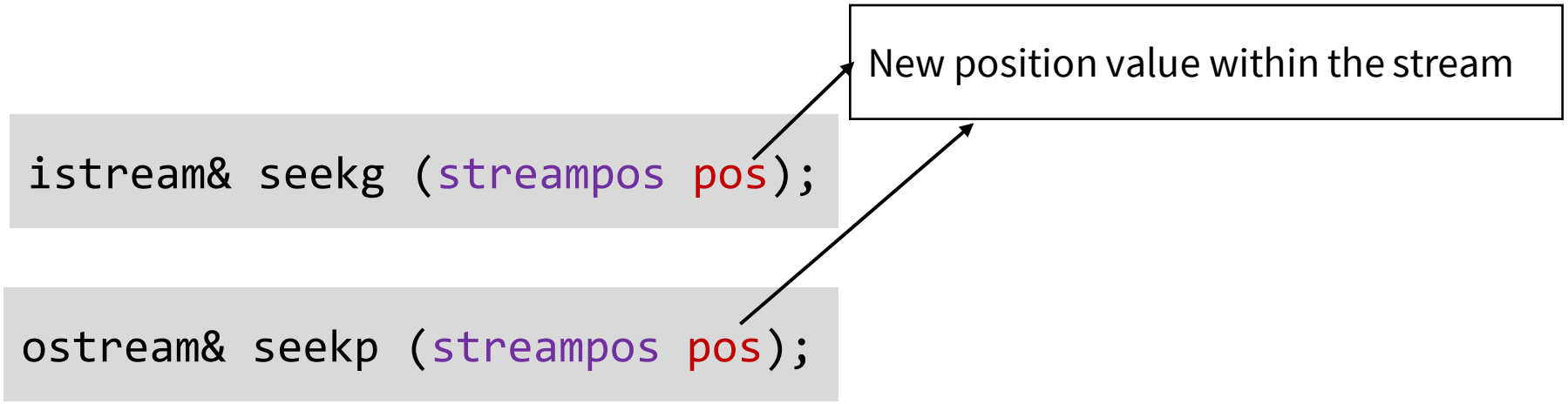
File position data type. Can be converted to integers

Random access

- To manipulate current position :
  - `seekg()` function sets the position of the next character to be extracted from the input stream
  - `seekp()` function sets the position where the next character is to be inserted into the output stream.
  - Two methods to set a position :
    - Set an **absolute** position
    - Set a **relative** position

Random access

- Absolute Position



Streampos:
File position data type. Can be converted to integers

Random access

- Relative Position

```
istream& seekg (streamoff off, ios_base::seekdir way);
```

Offset value relative to the way parameter

```
ostream& seekp (streamoff off, ios_base::seekdir way);
```

Object of type ios_base::seekdir. Can take any one of the following values

ios::beg	offset counted from the beginning of the stream
ios::cur	offset counted from the current position
ios::end	offset counted from the end of the stream

Dynamic Memory Allocation

Using malloc( ) and calloc( ) in C++

```
void *malloc(size_t size);
```

```
int main(){

    int *p;
    p= malloc(sizeof(int));
    *p=5;
    printf("%d", *p);
}
```

Valid in C. C allows void pointers to be implicitly converted to any other pointer type.

Invalid in C++: invalid conversion from ‘void*’ to ‘int*

Strong type checking in C++

```
void *calloc(size_t nmemb, size_t size);
```

```
int main(){

    int *p;
    p= (int*)malloc(sizeof(int));
    *p=5;
    printf("%d", *p);
}
```

Valid in C.

Valid in C++

Dynamic Memory Allocation

- C++ uses `new` and `delete` **operators** to create and destroy dynamic variables

- The `new` operator allocates memory for the variable and returns a pointer to it.

```
new datatype;
```

Allocates memory for datatype and returns a pointer to this memory location **(single object form)**

```
new datatype[an expression that evaluates to an integer];
```

Allocates memory for an **array** of datatype and returns a pointer to this memory location

Dynamic Memory Allocation

- The `delete` operator is used to return / release the memory that was allocated using the `new` operator.

```
delete ptr;
```

→ Deallocates memory pointed to by ptr**(single object form)**

```
delete [] ptr;
```

→ Deallocates memory block pointed to by ptr **(array form)**

Note:
new and delete call the constructors and destructors of objects respectively, whereas malloc, calloc, realloc and free do not.
new returns pointer to exact data type, while malloc() returns void *
On failure, malloc() returns NULL where as new throws bad_alloc exception.

Next Lecture

- **This week** (Tutorial Style Lecture):
  - Inheritance, vectors and file handling

- **Next Week :**
  - Containers (beyond vectors)
  - Templates