# CS 225 Data Structures

## Lab 1: Introduction to C++

**Klaus-Dieter Schewe**

**ZJU–UIUC Institute, Zhejiang University**

**International Campus, Haining, UIUC Building, B404**

**email: kd.schewe@intl.zju.edu.cn**

# 1 The Imperative Core of C++

## 1.1 Data Types

C++ is a strongly typed programming language, i.e. every object must have a well-specified type (or class)

Data types for **integers**:

- **short** (8-16 bits: -32768 . . . 32767)
- **int** (16 bits: -32768 . . . 32767)
- **long** (16-32 bits: -2147483648 . . . 2147483647)

**Operators** on integer data types: addition (+), subtraction (-), multiplication (*), division (/), modulo (%), increment (++), decrement (−)

**Comparison operators** are equality (==), inequality (!=) less than (<), less than or equal (<=), greater than (>), greater or equal (>=)

Note that **++i** differs from **i++** in the order of evaluation: **i++ / j** may give a different result than **++i / j**

# Natural Numbers, Booleans, Characters, Enumerations

All integer data types provide an **unsigned** version, e.g. in **unsigned int x**

These unsigned types are used to represent non-negative integers, i.e. natural numbers—thus, the range of **unsigned int** is 0 . . . 65535

Constants of type **long** are marked by suffix **L**, unsigned constants by suffix **U**

The data type for truth values is **bool** with only two values **true** and **false**

Comparison operators yield values of type **bool**

**Operators** on Booleans are negation (**!**), conjunction (**&&**) and disjunction (**| |**), as well as the comparison operators **==** and **!=**

The data type for characters is **char** with values in single quotes, e.g. **'A'**

**void** is data type with no values

An **enumeration type** is defined by a finite set of named constants, e.g.
**enum** traffic_light **{** red, yellow, green **}**

# Floating Point Numbers

Data types for **floating point numbers** are:

- **float** using 4 bytes, e.g. **3.142f** or **2.9979e8f**
- **double** using 8 bytes, e.g. **3.1415926535897932** or **2.997925e8**
- **long double** using 10 or 12 bytes, e.g. **0.5778143296e3L**

Operators and comparison operators are the same as for integers except %

There are automatic conversions between integers and floating point numbers

**Variable declarations** take the form **int x**, **double y = 2.997925e8**

For **assignments** use the operator = and the shortcuts +=, −=, *=, /=, %=

**Constant declarations** use the keyword **const** as in **const double pi = 3.141592653589793**

# 1.2 Control Structure

Every imperative language uses sequences of statements, so does C++; statements usually need a **;** at the end

**Compound statements** (aka **blocks**) are sequences of statements within braces: **{ . . . }**

Most importantly, the **scope** of a variable or constant declared inside a block is within the block

```
#include ⟨iostream⟩
using namespace std;
void main()
{ int x = 10;
        {   double x;
            x = 1.772;
            cout << x << "\n" ;        (prints double value 1.772)
        }
  cout << x << "\n" ;                  (prints int value 10)
}
```

# Branching Statements

**Branching statements** come in different forms:

- simple **if**-statements: **if (**⟨condition⟩**)** ⟨statement⟩, where the ⟨statement⟩ may be compound
- nested **if-else** statements:
  **if (**⟨condition$_1$⟩**)** ⟨statement$_1$⟩
  **else if (**⟨condition$_2$⟩**)** ⟨statement$_2$⟩
  ...
  **else** ⟨statement$_n$⟩
- case-by-case **switch**-statements:
  **switch (**⟨expression⟩**) {**
  **case** ⟨constant$_1$⟩**:** ⟨statement$_1$⟩**;**
  ...
  **case** ⟨constant$_n$⟩**:** ⟨statement$_n$⟩**;**
  **default:** ⟨last statement⟩**; }**

The statements ⟨statement$_i$⟩ in a **switch**-statement usually terminate with **break;**

If not, control drops through, which may prevent code being duplicated, but is disastrous, if the **break** is forgotten

# Loop Statements

**Loop statements** come in different forms:

- **while**-loops: **while (**⟨condition⟩**)** ⟨statement⟩, where the ⟨statement⟩ may be compound
- **for**-loops: **for (**⟨initialise⟩**;** ⟨condition⟩**;** ⟨change⟩**)** ⟨statement⟩

  Here ⟨initialise⟩ can be any statement that is executed first and only once

  The ⟨condition⟩ is checked in every iteration; if it becomes false, the loop will be terminated

  The ⟨change⟩ statement is executed after each iteration

  Any of ⟨initialise⟩, ⟨condition⟩ and ⟨change⟩ may be omitted, but not the **;**
- **do**-loops: **do** ⟨statement⟩ **while (**⟨condition⟩**);**

The **break** statement can also be used inside loops with the effect that the loop will be terminated

The **continue** statement can be used inside loops with the effect that the current iteration step will be terminated and the next iteration step will be started—note that in a **for**-loop the ⟨change⟩ statement is still executed in case of a **continue**

# Miscellaneous

The comma **,** can be used to separate a sequence of expressions, e.g. in initialisations **int** x = 10 **,** y = 4

This may be used also in **for**-loops for the ⟨initialise⟩ statement or the ⟨change⟩ statement, e.g. **for (int** $i = 0$ **,** $j = 0$ **;** $i < 10$ **&&** $j < 0$ **;** $++i$ **,** $++j$ **)** ...

A single semicolon **;** defines the **null statement**

The operator **?** permits to write a **conditional expression** in the form ⟨condition⟩ **?** ⟨result$_1$⟩ **:** ⟨result$_2$⟩

Command lines starting with #⟨keyword⟩ are not control statements, but rather instructions for the C++ preprocessor, e.g. for conditional compilation

The most important keywords for preprocessing are **include** and **define**

# 1.3 Functions

**Function declarations** take the form
**return_type function_name (type$_1$ identifier$_1$, . . . , type$_n$ identifier$_n$)**

**Function definitions** take the form
**return_type function_name (type$_1$ identifier$_1$, . . . , type$_n$ identifier$_n$)**
**{ function_body }**

Function declarations are usually placed in a separate header file

Functions with short body can be declared to be **inline**; then the function definition should also be in the header file; inline functions are merely an indication (for the compiler) to use the function body for code replacement rather than for implementation of function calls

Functions may be recursive, i.e. contains calls of themselves in the function body

Function bodies may contain declarations, but no declaration or definition of other functions (except in **main()**)

Unless the return type is **void**, the function body must contain a **return** statement, and the returned value must have (or be converted to) the return type

# Example

```
int factorial(int n) // calculates the factorial n!
{
  int result = 1 ;
  if (n > 0) {
  do {
         result *= n ;
         --n ;
     } while (n > 1);
  }
  else if (n < 0) { cout << "Error : negative argument = "
                  << n << "in factorial function\n" ; }
  return result;
}
```

We may prefer a recursive definition of a function computing factorials (exercise)

# Function Calls

A **function call** simply takes the form

**function_name(argument$_1$, . . . , argument$_n$)**

Such a function call in C++ usaes **call-by-value**, i.e. the arguments are copied and bound to the identifiers in the function declaration

Such a function call can be used like any other exression of the return type

A function declaration may contain arguments that are not used in the function body; then no argument identifier is required, but a function call must still provide an argument

A function declaration may provide **default values** for the identifiers (syntax is the same as for variable declarations with initialisation)

If default values are provided by the function definition, tailing arguments may be omitted in function calls (so the default values are used)

# Miscellaneous

The scope of any variable (including the identifiers in the head) declared inside a function is bound to the function

As a consequence, memory for such local variables is allocated when the function is invoked, and deallocated when control leaves the function

A function may also access variables outside the scope of the function by using the **scope resolution operator ::**, e.g. as in **::$x$ = ...**

Conversely, variables declared inside a function may be made persistent by using the keyword **static**, e.g. in **static double total_sum ;**

Static variables are only initialised once–by 0, if nothing else is specified—and retain their value after control leaves the function

Function names may be overloaded—the number of arguments or the argument types must be different

Function overloading may be useful, if "essentially the same" function is implemented (like a scalar product on vectors of different dimension), but should be avoided for other purposes

# 1.4 Pointers and Arrays

The address of an identifier is obtained by the **address-of** operator **&**, e.g. **&**$x$ retrieves the address of variable $x$

The **dereferencing operator** ***** retrieves the value at a given address (as in ***pt**—clearly, we have $^*\&x = x$

A **pointer** holds an address, usually the start address of some stored value—the special **null pointer** (**pt = 0**) does not point to any valid address

Pointers are declared by giving the type of the value stored at the address pointed to, e.g.

- **double *pt_x** declares a pointer with name pt_x, which directs to a location storing a double-precision floating-point number
- **int *pt_y** declares a pointer with name pt_y, which directs to a location storing an integer

As pointers contain addresses, they can be incremented, added and subtracted, e.g. in **++pt_x** or **pt_x += 4**

Pointers may point to pointers, e.g. in the declaration **int **pt** we have a pointer pt that points to a pointer to an integer

# Arrays

An **array** is a contiguous sequence of memory locations defined by a base address and a length

A declaration **type identifier[length]** defines an array with name identifier that stores as many values of the given type as indicated by the non-negative integer length, e.g.

- **double x[100]** defines an array of 100 double-precision floating point numbers
- **int *pt[10]** defines an array of 10 pointers to integers, whereas **int (*pt_x)[10]** defines a pointer pt_x to an array of 10 integers

For an array named $x$ the expression $x$**[i]** retrieves the $i$'th element of the array ($0 \leq i \leq \text{length} - 1$)—instead of $i$ we can use any expression that evaluates to an integer in the permissible range (we have $x[\text{i}] = *(\&x[0] + i)$)

The name of an array can be validly used as its base address, so an assignment **pt = x** can be used, where pt is a pointer, in particular for a generic pointer defined by **void *pt**

# Multi-Dimensional Arrays / Strings

Multi-dimensional arrays are defined and accessed analogously

For instance, **double x[4][7][2]** defines a $4 \times 7 \times 2$ array of doubles, and **x[2][3][1]** will retrieve the double at the address &x $+2 \cdot 7 \cdot 2 + 3 \cdot 2 + 1$

**Character strings** such as `"this is a string"` are treated as one-dimensional arrays of type **char**

This could be defined by **char message[]** = `"this is a string"`

However, in C++ such character strings always end with a special character '\0', so the array has always at least the size 1—this becomes relevant for string operations

Use **#include ⟨cstring⟩** to make string operations such as **strcpy()** and **strcat()** available

# Call by Reference

C++ also supports call-by-reference, which can be done in two ways:

- Using the **reference declarator &** the address of a memory location outside the scope of the function will be passed, as in **swap(int &x, int &y)**

  A function call looks the same as for call-by-value, e.j. **swap(i,j)** with variables i, j of type **int**

- Using a pointer as function argument, as in **void swap(int *pt_x, int *pt_y)**

  Then in a function call addresses must be passed, as in **swap(&i, &j)** with variables i, j of type **int**

As arrays de facto provide pointers, the second option can be used to handle arrays as funvction arguments, e.g. **double sum(double x[], int n)**—the argument **n** need not be the length of the array, and a length argument for the array argument is not permitted

A call can hand over any (one-dimensional) array, as in **sum(height,100)** or in **sum(height[10],38)** assuming a definition **double height[100]** or the like

# Examples

```
void swap(int *pt_x, int pt_y)
{  int temp;
   temp = *pt_x;
   *pt_x = *pt_y;
   *pt_y = temp;
}
```

```
void swap(int &x, int &y)
{  int temp;
   temp = x;
   x = y;
   y = temp;
}
```

```
int main()
{   int i = 10, j = 20;
    swap(&i, &j);
    cout << "i = " << i ;
    cout << "j = " << j ;
    cout << "\n";
    return(EXIT_SUCCESS);
}
```

```
int main()
{   int i = 10, j = 20;
    swap(i, j);
    cout << "i = " << i ;
    cout << "j = " << j ;
    cout << "\n";
    return(EXIT_SUCCESS);
}
```

# Multi-Dimensional Arrays and Functions as Arguments

Multi-dimensional arrays as function arguments are handled in the same way

However, all dimensions (except the first one) must be specified, as e.g. in **double trace(double x[] [5])**

We can declare pointer to a function, as e.g. in **double (*g)(int m)** (NOT **double *g(int m)**, which declares a function returning a pointer to a double)

Noter that in this case **g** contains the address of the function, where **g(...)** contains the value produced by the function on the given argument(s) **...**

In the same way pointer arguments to functions can be used in function declarations, e.g. **double sum(double (*g)(int m), int n)** declares a function **sum** that that two arguments, a pointer to a function on integers returning doubles plus an integer, the output of the function **sum** is a double

# Dynamic Memory Management

A C++ program has access to **free storage** (aka **heap**), so it can allocate and deallocate memory when needed

For allocation the operators **new** and **new[]** are used, e.g. **double *pt = new double;** allocates space for a new double, and
**double *pt_x = new double[100];** allocates space for an array of 100 doubles

In the former case a simultaneous initialisation is possible, e.g.
**double *pt = new double(0.459e4);**

The **new[]** operator can also be used to allocate space for an array of pointers, as e.g. in **double **pt_y = new double *[100];**

For deallocation the operators **delete** and **delete[]** are used, e.g. **delete pt;** or **delete[] pt_x;**

Dynamic memory management mostly makes sense for large data objects such as arrays or class objects, in particular for those data structures we will study in this course—for the standard types dynamic memory allocation and deallocation can usually be dispensed with

# 2 Object-Oriented Features of C++

## 2.1 Classes

Classes define **abstract data types** by means of data and functions members

Members can be **public** or **private**—only public members can be accessed from outside the scope of a class, whereas private members are encapsulated inside the class

Declarations of data and function members are done in the same way as in the imperative core

However, it is advisable to separate the definition of functions (including inline functions) from the class declaration using the **scope resolution operator ::**

**Objects** of a class are defined in the same way as for data types using **class_name object_name;**

Members of an object can be accessed by the the **class member access operator**, i.e. by using expressions of the form **object_name.member_name**

# Example

```
class int_list {
public:
    int_list(int size = 20);
    int get_item(int index);
    void set_item(int newitem);
    void insert(int item);
    int length;
private:
    int maxsize;
    int *data; }

inline int_list::int_list(int size)
{   maxsize = size;
    data = new int[size];
    last = -1; }
int int_list::get_item(int index)
{   if (0 <= index && index <= length)
        { return data[index]; }
```

# Example (cont.)

```
    else
        { cout << "list index " << index << " out of range\n";
          return(EXIT_FAILURE); } }
void int_list::set_item(int index, int newitem)
{   if (0 <= index && index < length)
        { data[index] = newitem;
          return; }
    else
        { cout << "list index " << index << " out of range\n";
          return(EXIT_FAILURE); } }
void int_list::insert(int item)
{   if (length < maxsize)
        { data[length + 1] = item;
          ++length; }
    else
        { cout << "list is already filled\n";
          return(EXIT_FAILURE); } }
```

# Classes and Functions

Objects of a class can be arguments of functions, and can be returned by functions

**Example:**

```
int sum(int_list list)
{ int sum = 0;
  for (i=0; i <= list.length; ++i)
  { if (sample.getitem(i) % 2 != 0) sum+= i; }
}
int main()
{ int_list sample = int_list(20);
  for (i=0; i < 20; ++i) {sample.insert(3*i*i-i+1);}
  return sum(sample);
}
```

In order to avoid redundant copying it is advisable to use a reference argument in such a function; otherwise a new object might be created by the function and copied(!) to another new object in the **return** statement

# Access Operators

The c ommon access to members uses the class member access operator **object_name.member_name**

If pt is a pointer to an object of some class, then **(*pt).member** access the (public) class member that is declared in the class; this can be abbreviated by **pt->member** using another **class member access operator ->**

Using the scope resolution operator, we may use **type_name class_name::*pointer** to define a pointer to a class member of the given type, and an assignment of the form **pointer = &class_name::member** then creates a pointer to the specified class member

Then **object.*pointer** or **(object.*pointer)(...)** accesses the member of an object pointed to; **.*** is the **pointer-to-member access operator**

A second **pointer-to-member access operator** is **->***, where **pt_s->*pt** is a shortcut for **(*pt_s).*pt**

# Miscellaneous

Class members can be declared to be **static**, as in **static int total;**

Static data members are by default initialised as 0, but a different initialisation is possible (though not within the class declaration, but using
**type class_name::member_name = value**

Static members are shared by all objects of a class; the same applies to static member functions

A call of a static member function of a class uses the class name rather than the object name

Objects of a class can also be declared as constant using the preceding keyword **const**

Functions on such constant objects can only be executed, if they also have been declared as constant (not modifying the members); in this case the keyword **const** is added at the end of the function declaration

# Friend Functions and Classes

A function can be declared to be a **friend** of a class, e.g. in
**friend int get_item(int index);**

A friend function can access the members of the class, but it is not a member of the class itself

More generally, we can declare a class to be a friend, e.g. in **friend class graph**

Then all functions of the friend class have access to all members of the given class

It is therefore possible to define **private classes** with only private members, but with certain classes as friends

Then only the friend classes can access the members of a private class; otherwise a private class would be useless

Note that the friend concept is not transitive

# Constructors and Destructors

A member function with the same name as the class itself is a **constructor function**

A class may have more than one constructor function, but the argument list must differ

For instance, **int_list(int size = 20)** in our class example above declares a constructor for the class int_list

A constructor is used to initialise a new object (as in our previous example)

We could define another constructor by **int_list(int_list &$\ell$)** using the members of a specific object as the default

A **destructor function** has a name **~class_name**

A destructor has no return type, no arguments, and cannot be static

# 2.2 Inheritance in C++

In C++ a class may be declared as a **derived class** using **class derived_class : [public|private] base_class$_1$ , . . . , [public|private] base_class$_n$ { . . . }**

There must be one or more previously defined classes from which the class can be derived; these classes can also be derived from other classes

In case $n = 1$ we talk of **simple inheritance**, otherwise of **multiple inheritance**

A derived class defines its members in the same way as any other class, but in addition has all data and function members of its base class(es); it **inherits** these members (except constructors and destructors) from the base class(es)

A base class can be **private** or **public** for a derived class (private is the default)

If a base class is private for a derived class, all inherited members are private

If a base class is public for a derived class, all inherited public members become public, but private members remain private to the base class

# Dominance

In C++ the inheritance concept merely extends the class definition(s) by new members (**inclusion polymorphism**), but the same names may be overloaded (**ad-hoc polymorphism**)

If member names are re-used, any ambiguity is resolved by the member definition in the derived class being **dominant**, i.e. the class member access operator always refers to the member definition for the derived class

If member names are re-used, the member defined for a base class can still be accessed by using the scope resolution operator **::**, as in **object.base_class::member**

Here **object** is an object of a class derived from **base_class**, and **member** was defined for this base class

If member names are not re-used, it may also be necessary to use the class resolution **::**, if there is more than one base class, and member names are not unique

# Ad-hoc Polymorphism

The overloading of function names with different definitions in derived and base classes is called **ad-hoc polymorphism** in type theory

It may become necessary that function members of different classes derived from a common base class are needed, but the class of an object is only known at run-time

For this C++ allows us to declare a function member (but not a constructor or destructor) to be virtual using the keyword **virtual**, which must precede the function declaration in the base class (and only there)

It is also possible not to have a function definition for the base class (only a declaration), if such a function would only make sense for specific derived classes

The function declaration in this case takes the form

**virtual type function_name(. . . arguments . . . ) = 0**

indicating that the function is a **pure virtual function**, i.e. it is undefined

A class with with at least one such undefined member functions is called an **abstract class**

# Access Specifiers

In addition to **private** and **public** C++ provides another **access specifier**: **protected**

Access specifiers are used for the declarations of class members and in the list of base class(es) of a derived class

The following rules govern the combination of these access specifiers:

- If a base class is private, all its members are private in derived class
- If a base class is public, all its members have the same access specifier in the base and any derived class
- If a base class is protected, its public and protected members are protected in any derived class, while private members remain private

# Miscellaneous

As derived classes can be defined using another derived class as its base class, we can define deep class hierarchies (note that circularity is not permitted)

With multiple inheritance it is possible that a class becomes an indirect base class in multiple ways, i.e. an object may contain multiple copies of members from such an indirect base class

This can be avoided by making base classes virtual using the keyword **virtual** preceding the access specifier

Constructors and destructors cannot be inherited, but a definition of a constructor of a derived class must call the constructor of the base class:
**derived_class_constructor(...) : base_class_constructor(...){ ... }**

In case of multiple inheritance a list of base class constructors can be called; this list may even contain constructors of virtual indirect base classes

# 2.3 Parametric Polymorphism in C++

Strong typing requires that all arguments of a function are typed and a return type must be specified

In order to enable the definition of functions that depend only on restricted properties of a type, C++ provides the concept of a **function template**, which permits data types and classes as parameters

In type theory this is called **parametric polymorphism**

A template argument list is a list of class/type variables each preceded by the keyword **class**, i.e. the syntax is $\langle$**class** $T_1$**, . . . ,** **class** $T_n\rangle$

A function template is then declared by **template template_argument_list function_declaration**, where in the function_declaration the class/type variables are treated just like any other type name

A function template definition extends such a declaration by a function body; again, the class/type variables are treated just like any other type name

# Function Template Calls

**Example.** Define a min function, which only requires that a total order $\leq$ is defined for a type

```
template<class T> T min(T m, T n)
  { return ((m <= n) ? m : n) }
```

A function template can be called like any other function, e.g. `min(x,y)`

It is not always possible to omit the concrete type replacing the parameter in the template

For instance, using `min(x,y)` for an integer x and a double y could return either a double (i.e. treat x as double) or an integer (i.e. coerce y to an integer)

Such conflicts can be avoided by explicitly giving the argument type, e.g. we could use `min<double>(x,y)` or `min<int>(x,y)`

# Class Templates

The template concept is not restricted to functions; it can as well be used for classes

In this case the class template declaration becomes
**template template_argument_list class_declaration**, where in the class_declaration the class/type variables are treated just like any other type name

Within the class template declarations and definitions there is no need to qualify the parameterised class by the type/class parameters

However, the definition of class members outside the scope of the class template declaration requires these parameters, i.e. the prefix
**template template_argument_list** is needed

Furthermore, the type/class parameters must be used in the function names

# 3 Miscellaneous Topics

**File Organisation.** A program file containing **main()** may (conditionally or unconditionally) include other files, in particular header files and libraries

Declarations of functions and classes are usually kept in header files (extension .h), while function definitions (except inline functions), class member definitions, etc. are placed in the program file (extension .cxx or .cpp depending on the environment)

**Namespaces.** In order to prevent name clashes (from inclusion of multiple header files) C++ permits the definition of **namespaces**

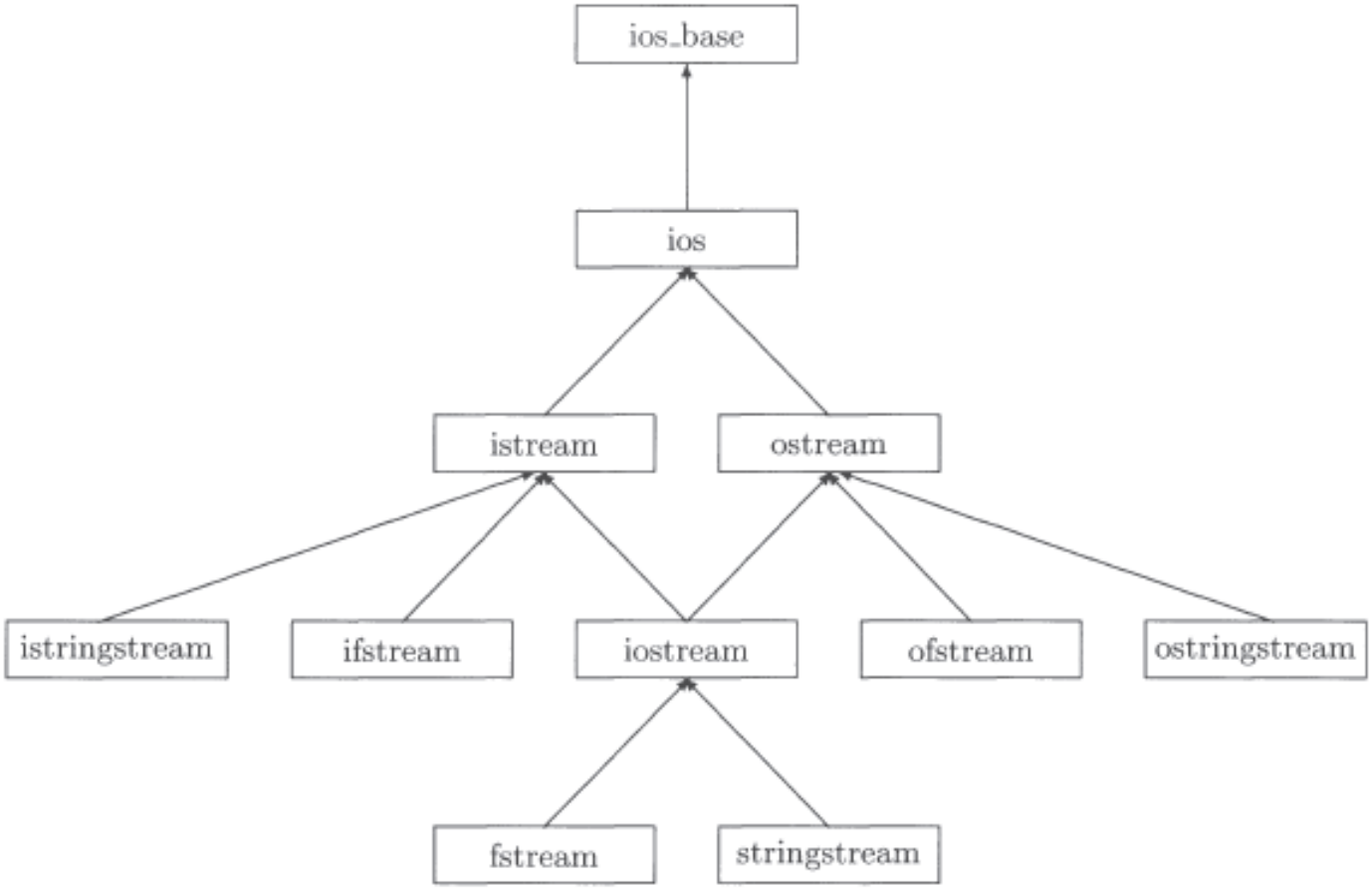
Names declared in some program or header file can be added to a named namespace

With **using namespace** or **using namespace::name** all or selected names from a namespace can be made accessible in a program or header file

# Input and Output

Input and output are supposed to be taken from or directed to streams, respectively

For these a hierarchy of library header files is available



# I/O Streams

For instance, ⟨iostream⟩ defines **cin** and **cout**

Input is taken from the input stream using **cin >> ...**, whereas output flows towards the output stream, i.e. we use **cout << ...**

The list of I/O library header files is the following:

| Header | Purpose |
|---|---|
| `<iosfwd>` | forward declarations |
| `<iostream>` | standard iostream objects |
| `<ios>` | iostreams base classes |
| `<streambuf>` | stream buffers |
| `<istream>` | input stream template |
| `<ostream>` | output stream template |
| `<iomanip>` | formatting and manipulators |
| `<sstream>` | string streams |
| `<fstream>` | file streams |
| `<cstdio>` | C-style I/O |

# Files for Input and Output

For input from a file we need to **#include ifstream**, we output to a file analogously **#include ofstream**

With ifstream we can declare a file, e.g. "example_input.dat" to be used for the input stream:

**ifstream input_file("example_input.dat");**

Then the declared **input_file** take on the role of cin, i.e. we can read input from the file using

**input_file >> x**

Analogously, with ofstream we can declare a file, e.g. "example_output.dat" to be used for the input stream:

**ifstream output_file("example_output.dat");**

Then the declared **output_file** take on the role of cout, i.e. we can read input from the file using

**output_file << x**

For input and output to file there are several options how to open the file, how to handle errors, how to write or read data, etc. that need to be specified with the declaration of I/O files (see the C++ documentation or a C++ textbook)