PHY1610H – Scientific Computing for Physicists

Daniel Gruner, Ramses van Zon,
Marcelo Ponce, Scott Northrup

SciNet HPC Consortium/Physics Department
University of Toronto

Winter 2017

About the course

About the course

- Whole-term graduate course
- Prerequisite: some programming experience (C, C++, Fortran, Java, C#, ...).
- Will use C++
- Topics include: Scientific computing and programming skills, optimization, parallel programming

About the course

1. Scientific Software Development: Jan 2017

   C++, version control, automated builds, modular programming, testing, debugging

2. Numerical Tools for Physical Scientists: Feb/Mar 2017

   Modelling, floating point, random numbers and Monte Carlo, ODE & MD, linear algebra, PDE, FFT

3. High Performance Scientific Computing: Mar/Apr 2017

   Profiling, Optimization, Parallel Programming (OpenMP and MPI)

You can take it as a whole, or, if you're in chem, astro, or physics, as mini-courses.

But let us know!

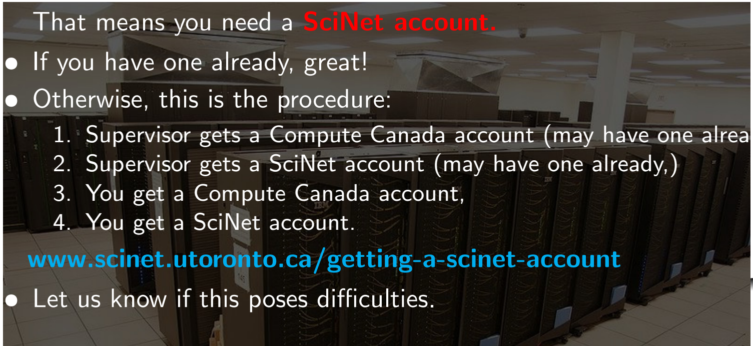
If you're not taking the course for credit, you can still take it for a SciNet Certificate.

About us

- Your instructors are Computational Science Analysts from SciNet:
  Ramses van Zon, Marcelo Ponce, Scott Northrup
- SciNet is a consortium of UofT and research hospitals, that hosts and supports Canada's largest academic supercomputers.
- We also do a lot of teaching (Bash, Python, R, Fortran, C++, CUDA, databases, machine learning, parallel programming, visualization, ...)
- SciNet is part of Compute Canada, and is open to all academic researchers in Canada.

Accounts, homework, ...

- At the beginning of the course, you can do the homework assignments on your own computer, provided it has
  - A unix-like environment with the **GNU compiler suite** (i.e. **g++**) and **make** (Linux: done. Windows: **cygwin**. Mac: one of the GNU compiler suites from **hpc.sourceforge.net**).
  - The version control software called **'git'** installed.
- Towards the third part of this course, you'll be working on one of our supercomputers.



That means you need a **SciNet account.**

- If you have one already, great!
- Otherwise, this is the procedure:
  1. Supervisor gets a Compute Canada account (may have one alrea
  2. Supervisor gets a SciNet account (may have one already,)
  3. You get a Compute Canada account,
  4. You get a SciNet account.

  www.scinet.utoronto.ca/getting-a-scinet-account
- Let us know if this poses difficulties.

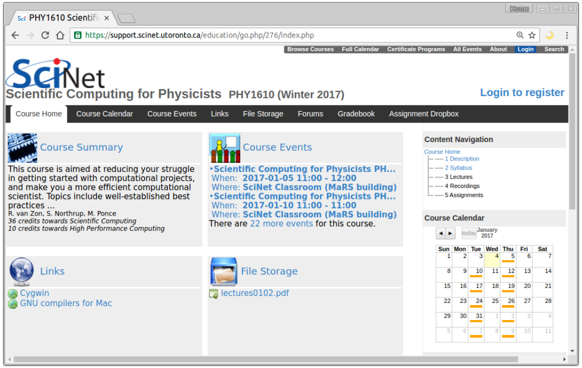
Course website

https://support.scinet.utoronto.ca/education/go.php/276/index.php

- Lectures (+Recordings)
- Assignments
- Forum
- ...

Near-weekly assignments given on Tuesdays, posted on the site.

To be able to submit homework and get course emails, you to be able to login to the site (use your SciNet account if you have one).


PHY1610 Scientifi ×

https://support.scinet.utoronto.ca/education/go.php/276/index.php

Browse Courses | Full Calendar | Certificate Programs | All Events | About | Login | Search

SciNet

Scientific Computing for Physicists PHY1610 (Winter 2017)

Login to register

Course Home | Course Calendar | Course Events | Links | File Storage | Forums | Gradebook | Assignment Dropbox

Course Summary

This course is aimed at reducing your struggle in getting started with computational projects, and make you a more efficient computational scientist. Topics include well-established best practices ...

R. van Zon, S. Northrup, M. Ponce

36 credits towards Scientific Computing

10 credits towards High Performance Computing

Course Events

•Scientific Computing for Physicists PH...
When: 2017-01-05 11:00 - 12:00
Where: SciNet Classroom (MaRS building)

•Scientific Computing for Physicists PH...
When: 2017-01-10 11:00 - 11:00
Where: SciNet Classroom (MaRS building)

There are 22 more events for this course.

Links

Cygwin
GNU compilers for Mac

File Storage

lectures0102.pdf

Content Navigation

Course Home
- 1 Description
- 2 Syllabus
- 3 Lectures
- 4 Recordings
- 5 Assignments

Course Calendar

today January 2017

Sun	Mon	Tue	Wed	Thu	Fri	Sat
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4
5	6	7	8	9	10	11

If you are going to take the course for (physics) credit, make sure you are also signed up for the course in ACORN.

Grading scheme

- **Near-weekly programming assignments** posted on the website.
- These assignments are **due the next week.**
- The average of the assignments will make up your grade.
- All sets of homework need to be handed in for a passing grade.

Penalty policy

- Homework may be submitted up to 1 week after the due date, at a penalty of $\frac{1}{2}$ point per day, out of the 10 points per homework.
  Deviations of this rule will only be considered, on a case-by-case basis, in exceptional circumstances (i.e., not "I was busy").
- If, due to exceptional circumstances, an assignment was missed, a make-up assignment on a topic of the instructors' choice can be given at the end of the course.

Office hours, email, lecture broadcasts

Office hours

For the duration of the course, office hours will be on Wednesdays from 2:00 pm to 3:00 pm at:
SciNet Offices
MaRS West Tower, 11th floor
661 University Avenue
Toronto, ON

Questions/comments/concerns/etc. about the course?

For questions regarding the course, use the email **courses@scinet.utoronto.ca**
All instructors are on this email list.

Have to miss a class?

- Lectures are recorded and posted on the site afterwards (often towards the end of the day).
- We're exploring the possibility for broadcasting the lectures as well. Instructions with the link to be broadcast are posted in the chat on the course website just before the lecture starts.

Course Outline

Part 1 - Scientific Software Development

Lecture 1	C++ intro
Lecture 2	More C++
Lecture 3	Modular programming
Lecture 4	Make
Lecture 5	Arrays and other data structures
Lecture 6	Objects and debugging
Lecture 7	Version control with git
Lecture 8	Unit testing

Part 2 - Numerical Tools for Physicists

Lecture 9	Numerical errors
Lecture 10	ODEs and Molecular Dynamics
Lecture 11	Partial differental equations
Lecture 12	Root finding and optimization
Lecture 13	Linear algebra review
Lecture 14	Linear algebra libraries
Lecture 15	Fast Fourier transforms
Lecture 16	Monte Carlo and random numbers

Part 3 - High Performance Computing

Lecture 17	Profiling tools
Lecture 18	Serial performance
Lecture 19	Intro to parallel computing
Lecture 20	Batch processing
Lecture 21	Shared memory programming 1
Lecture 22	Shared memory programming 2
Lecture 23	Distributed parallel programming 1
Lecture 24	Distributed parallel programming 2

Part I

Introduction to Scientific Software Development

Programming

Programming

- We program to have the computer perform a number of similar computations or data manipulations.
- A program specifies the actions that the computer should take, as well as (restrictions on) the order in which they should be taken.
- Each action will have a net effect on the program's "state".
- There is limited set of predefined actions, in terms of which we must express all other actions: that is programming.

Programming

- A common pattern of actions to achieve a specific net effect (*computation*) is an **algorithm**.
- A **function**, **procedure**, or **subroutine** is a specification of actions that can be used as a newly defined action.
- A **program** is a function that can be executed.
- Programs may accept some external data as **input** and produce data as **output**.

Program State

- Program state is stored in memory.
- At least part of the state is made up of the program's **variables**.
- Variables are values that are assigned to a **variable name**.
- This variable name is associated with a portion of **memory** that holds the variable's value.

Control structures

- Some actions could be done conditionally on the state of the program and external input. **Conditional control structures** perform a different actions depending on whether a certain assertion of the state of the system is true.
- Repetition of a set of actions: **loops**.

Some ideas were taken from:

"A Short Introduction to the Art of Programming" (E. W. Dijkstra, 1971)
http://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD316.3.html

Why C++?

Advantages

- High performance
- Low-level programming
- Ubiquitous and standardized
- Useful libraries
- Modular design

Disadvantages

- Labour intensive, error prone
- High-level programming up to you
- Non-interactive
- Things like graphics can be hard
- Beware of performance pitfalls

For e.g. Python, many advantages and disadvantages are reversed.

No free lunch:

You can program poorly and inefficiently in any language.

C++ Introduction

C++ Introduction

- C/C++ are **compiled** languages: their basic 'actions' are to be compiled into a set of basic 'native' instructions that the processor can execute.
- C was designed for (unix) system programming.
- C has a very small base.
- Most functionality is in (standard) libraries.
- C++ is almost a superset of C.
- For definiteness sake, let's say we use the **C++11** standard.

C++ Introduction: Basic C++ program

```
// hw.cc - prints "Hello world."
#include <iostream> // to define input/output routines
#include <string> // to define strings
int main() // called first
{ // braces delimit a code block
    std::string message = "Hello world."; // a variable
    std::cout << message     // print to console out
              << std::endl; // end of line
    // semicolon after statement
    return 0;
    // return value to shell
}
```

Command line compilation:

```
$ g++ -std=c++11 -o hw hw.cc -g -O2
$ ./hw
Hello world.
$ echo $?
0
```

C++ Intro: Basic syntax aspects

- Other C++ files can be included with the `#include` directive.
- Each executable statement or declaration ends with a semicolon.
- Curly braces delimit a code block.
- When declaring a variable or function to be of a certain type, the type is specified before the variable or function name.
- The value to be given back by a function is specified by the `return` statement, which exits the function.
- Comments can be added using the double slashes //.

C++ Intro: Variables

- Variables are named pieces of data stored in the computers memory.
- In C/C++ variables have a particular type, and are defined as having a particular type.
- The data stored in a variable can be manipulated (e.g. through assignment, addition, . . .).
- E.g. the same name cannot be used to refer to both an integer and a string, and a string cannot be stored in a variable that has been declared as an integer.

C++ Intro: Variable definition

```
type name [= value];
```

Here, *type* may be a:

* floating point type:
  `float`, `double`, `long double`, `std::complex<float>`, ...
* integer type:
  [`unsigned`] `short`, `int`, `long`, `long long`
* character or string of characters:
  `char`, `char*`, `std::string`
* boolean:
  `bool`
* array, pointer
* class, structure, enumerated type, union

Non-initialized variables are not 0, but have random values!

C++ Intro: Examples of Variables

```
int a;
int b;
a = 4;
b = a + 2;
```

```
float f = 4.0f;
double d = 4.0;
d += f;
```

```
char* str = "Hello There!";
```

```
bool itis2016 = false;
```

C++ Intro: Functions

Function = a piece of code that can be reused

A function has:

1. a name
2. a set of arguments of specific type
3. and returns a value of some specific type

These three properties are called the function's **signature**.

- To write a piece of code that **calls** the function, we only need to know its signature; For this, one places a **function declaration** before the piece of code that is to use the function.
- The actual code (**function definition**) can be in a different file or in a library.
- Function declarations of a library are in header files that are **included**.

C++ Intro: Functions

- Function declaration (prototype/signature)

```
returntype name(argument-spec);
```

argument-spec = comma separated list of variable definitions

returntype could be `void` .

e.g.: `int main(int argc, char ** argv);`

- Function definition

```
returntype name(argument-spec) {
    statements
    return expression-of-type-returntype;
}
```

- Function call

```
var=name(argument-list);
f(name(argument-list));
name(argument-list);
```

argument-list = comma separated list of values

C++ Intro: Functions, an example

```
// external function prototypes
#include <cmath>
// function prototype
double arithmetic_mean(double a, double b);
// main function to get called when program starts:
int main()
{
   double x = 16.3;
   double y = 102.4;
   double z;
   z = arithmetic_mean(x,y);
   return (int)(z);
}
// function definition
double arithmetic_mean(double a, double b)
{
   return sqrt(a*b);
}
```

C++ Intro: Pass by value or by reference

Passing function arguments by reference

```
// passval.cc
void inc(int i) {
   i = i+1;
}
int main() {
   int j = 10;
   inc(j);
   return j;
}
```

```
$ g++ -std=c++11 -o passval passval.cc
$./passval
$ echo $?
10
$ █
```

- `j` is set to 10.
- `j` is passed to `inc`,
- where it is copies into a variable called `i`.
- `i` is increased by one,
- but the original `j` is not changed.

C++ Intro: Pass by value or by reference

Passing function arguments by reference

```
// passref.cc
void inc(int &i){
   i = i+1;
}
int main() {
   int j = 10;
   inc(j);
   return j;
}
```

```
$ g++ -std=c++11 -o passref passref.cc
$ ./passref
$ echo $?
11
$ 
```

- `j` is set to 10.
- `j` is passed to `inc`,
- where it referred to as `i` (but it's still `j`).
- `i` is increased by one,
- Because `i` is just another name for `j`, `j` reflects this increase.

C++ operators

Arithmetic

`a+b`	Add a and b
`a-b`	Subtract a and b
`a*b`	Multiply a and b
`a/b`	Divide a and b
`a%b`	Remainder of a over b

Logic

`a==b`	a equals b
`a!=b`	a does not equal b
`!a`	a is not true (also: `not a`)
`a&&b`	both a and b true (also: `a and b`)
`a||b`	either a or b is true (also: `a or b`)

Assignment

`a=b`	Assign an expression b to the variable b
`a+=b`	Add b to a (result stored in a)
`a-=b`	Subtract b from a (result stored in a)
`a*=b`	Multiply a with b (result stored in a)
`a/=b`	Divide a by b (result stored in a)
`a++`	Increase value of a by one
`a--`	Decrease value of a by one

What is `1/4`?

```
1/4 = 0
```

Why?

- Literal expressions, such as
  `"Hi", 0, 1.2e-4, 2.4f, 0xff, true`
  have types, just as variables do.
- The result-type of an operator depends on that of the operands.
- In “1/4” both operands are integers
- Hence, the result of 1/4 is the integer part of the division, which is **0**.

To fix it, we need to be able to convert between types. In C/C++ this is called casting.

Casting one numeric type into another

Treat the type as a function.
E.g.

```
// 1over4.cc
#include <iostream>
int main() {
    int a = 1;
    int b = 4;
    int c = a/b;
    float d = float(a)/float(b);
    std::cout
        << c << " "
        << d << " "
        << int(d) << std::endl;
}
```

```
$ g++ -std=c++11 1over4.cc -o 1over4
$ ./1over4
0 0.25 0
```

Automatic Casting

If an expression expects a variable or literal of a certain type, but it receives another, C++ may be able to convert it automatically.
E.g.

```
1.0/4
```

is equal to

```
1.0/4.0
```

The expression may be a function call too, so that in

```
int unchanged(int i) {
    return i;
}
int main() {
    return unchanged(2.3);
}
```

the argument 2.3 gets converted to an int first, and then passed to the function unchanged, so the returned value is 2.

C++ Intro: Loops

- In scientific computing, we often want to do the same thing for all points on a grid, or for every piece of experimental data, etc.
- If the grid points or data points are numbers, this means we consecutively want to consider the first point, do something with it, then the second point, do something with it, etc., until we run out of points.
- That's called a loop, because the same 'do something' is executed again and again for different cases.

C++ Intro: Loops

Two forms:

```
for (initialization; condition; increment) {
    statements
}
```

```
while (condition) {
    statements
}
```

You can use `break` to exit the loop.

C++ Intro: Loop example

```
// count.cc
#include <iostream>
int main() {
   for (int i=1; i<=10; i++)
      std::cout << i << " ";
      // look, no braces!
   std::cout << std::endl;
}
```

```
$ g++ -std=c++11 -o count count.cc -O2
$ ./count
1 2 3 4 5 5 6 7 8 9 10
$ █
```

C++ Intro: Pointers

- Pointers are essentially memory addresses of variables.
- For each type of variable *type*, there is a pointer type *type** that can hold an address of such a variable.
- Useful in arrays, linked lists, binary trees, ...
- Null pointer, denoted by nullptr, points to nowhere.

Definition:

```
type *name;
```

Assignment ("address-of"):

```
name = &variable-of-type;
```

Deferencing ("content-at"):

```
variable-of-type = *name;
```

C++ Intro: Pointers

```
// ptr.cc
#include <iostream>
int main() {
   int a = 7, b = 5;
   int *ptr = &a;
   a = 13;
   b = *ptr;
   std::cout<< "b=" << b << std::endl;
}
```

```
$ g++ -std=c++11 -o ptr ptr.cc
$ ./ptr
b=13
$ █
```

- Two integer variables a and b are initialized with values 7 and 5, respectively.
- A pointer `ptr` is set to point to a.
- a is changed to 13.
- b is changed to be whatever `ptr` is pointing to.
- Because `ptr` was pointing at a, b ended up being changed to 13 to.

C++ Intro: Automatic arrays

```
type name[number];
```

- *name* is equivalent to a pointer to the first element.
- Usage: *name*`[i]`. Equivalent to `*(`*name*`+i)`.
  This is really a just a different way to dereference a pointer.
- C/C++ arrays are zero-based.

C++ Intro: Automatic arrays

Example

```
// autoarr.cc
#include <iostream>
int main() {
    int a[6]={2,3,4,6,8,2};
    int sum=0;
    for (int i=0;i<6;i++)
        sum += a[i];
    std::cout << sum << std::endl;
}
```

B A D !!

(in general)

```
$ g++ -std=c++11 -o autoarr autoarr.cc -O2
$ ./autoarr
25
$ █
```

Why bad?

- C standard only says at least one array of at least 65535 bytes.
- In practice, limit is set by compiler and OS stack size.

C++ Intro: Dynamically allocated array

A dynamically allocated arrays is defined as a pointer to memory:

```
type *name;
```

Allocated using the keyword `new` :

```
name = new type [number];
```

Deallocated by a function call:

```
delete [] name;
```

- Usage of these arrays is the same as for automatic arrays.
- Can access all available memory.
- Can control when memory is gived back.
- Unfortunately, no multi-dimensional version in the standard.

Array allocation - Improved version

Example

```
// dynarr.cc
#include <iostream>
int main() {
    int *a = new int [6];
    a[0] = 2; a[1] = 3; a[2] = 4;
    a[3] = 6; a[4] = 8; a[5] = 2;
    int sum=0;
    for (int i=0;i<6;i++)
        sum += a[i];
    std::cout << sum << std::endl;
    delete [] a;
}
```

```
$ g++ -std=c++11 -o dynarr dynarr.cc -O2
$ ./dynarr
25
$ █
```

C++ Intro: Dynamically allocated arrays

Example

```
// dyna.cc
#include <iostream>
void printarr(int n, int *a)
{
   for (int i=0;i<n;i++)
      std::cout << a[i] << " ";
   std::cout << std::endl;
}
int main()
{
   int n = 100;
   int *b = new int [n];
   for (int i=0;i<n;i++)
      b[i]=i*i;
   printarr(n,b);
   delete [] b;
   return 0;
}
```

```
$ g++ -std=c++11 -o dyna dyna.cc -O2
$ ./dyna
0 1 4 9 16 25 36 49 64 81 100 121 144 169
196 225 256 289 324 361 400 441 484 529 576
625 676 729 784 841 900 961 1024 1089 1156
1225 1296 1369 1444 1521 1600 1681 1764 1849
1936 2025 2116 2209 2304 2401 2500 2601 2704
2809 2916 3025 3136 3249 3364 3481 3600 3721
3844 3969 4096 4225 4356 4489 4624 4761 4900
5041 5184 5329 5476 5625 5776 5929 6084 6241
6400 6561 6724 6889 7056 7225 7396 7569 7744
7921 8100 8281 8464 8649 8836 9025 9216 9409
9604 9801
```

Dynamic allocation of single variables

One can also dynamically allocate a single variable:

```
double * a = new double;
*a = 4;
std::cout << a << std::endl;
delete a;
```

Note the absence of `[]`.

You might use this in more dynamic data structures.

C++ Intro: Conditionals

```
if (condition) {
    statements
} else if (other condition) {
    statements
} else {
    statements
}
```

Example

```
int main(){
    int n = 20;
    int *b = new int [n];
    if (b == nullptr)
        return 1; //error
    else {
        for (int i=0;i<n;i++)
            b[i] = i*i;
        printarr(n,b);
        delete [] b;
    }
}
```

```
$ g++ -std=c++11 -o ifm ifm.cc -O2
$ ./ifm
0 1 4 9 16 25 36 49 64 81 100
121 144 169 196 225 256 289
324 361
$ █
```

C++ Intro: Const

A type modifier

- const is a type modifier.
- It means the value of that type is fixed.
- Useful for constants, e.g.

```
const int arraySize = 1024;
```

- Useful to show read-only arguments to functions:

```
int f( const Type &in, Type &out );
```

- const is contageous!
- Now everything has to be “const correct”.

C++ Intro: Objects

Object oriented programming (OOP)

- **Non-OOP:** functions and data accessible from everywhere.
- **OOP:** Data and functions (**methods**) together in an **object**. Implementation details **hidden**.

What are classes?

- Classes are to objects what types are to variables.
- Using a class, one can create one or more **instances** of it, called **objects**:

```
classname objectname(arguments);
```

C++ Intro: Classes and objects

Syntax:

```
classname objectname(arguments);
```

Usage

- Different from regular variables are the possibility of argument, supplied to **construct** the object.
- An object has **members** (fields) and **member functions** (methods), which are accessed using the “.” notation.

```
object.field;
object.method(arguments);
```

C++ Intro: Classes and objects

Example (member function/method)

```
#include <string>
std::string s("Hello");
int stringlen=s.size();
```

Example (member/field)

```
#include <utility>
std::pair<int,float> p(1, 0.314e01);
int   int_of_pair   = p.first;
float float_of_pair = p.second;
```

C++ Intro: Templates

Templates

- Some algorithms are type-independent, and can be expressed with the same code, except for a change in type.
- Using *generic programming*, you write this code once, with a generic type placeholder. Versions of this code for specific **types** are **instantiated** by the compiler when needed.
- In C++, generic programming uses **templates**.
- Many templated functions and classes in the standard library.

C++ Intro: Template functions

```
int sqrit(int x) { return x*x; }
double sqrit(double x) { return x*x; }
```

- Imagine you had these two functions to compute the square of a number, one for `int` s and one for `double` s: Note that they have the same name: this is fine in C++, the type of arguments in a call will determine which is used.
- We would have to write one of these functions for each type.
- Templates allow us to write the generic function just once:

```
template<typename TYPE>
TYPE sqrit(TYPE x) { return x*x; }
```

- Wherever the compiler sees a function call `sqrit(...)` it generates the right function.
- Where there's ambiguity, you can be more explicit, e.g. `sqrit<double>(3);`

C++ Intro: Class Templates

Usage

- To create an object from a template class *templateclass*:

```
templateclass<type> object(arguments);
```

Examples:

```
std::complex<float> z; //single precision complex number
std::vector<int> i(20);//array of 20 integers
```

C++ Intro: Libraries

Usage

- Put an include line in the source code, e.g.

```
#include <iostream>
#include "mpi.h"
```

- Include the libraries at link time using `-l[libname]`.
  Implicit for the standard libraries.

Common standard libraries (Standard Template Library)

- string: character strings
- iostream: input/output, e.g., `cin` and `cout`
- fstream: file input/output, e.g., `ifstream` and `ofstream`
- containers: vector, complex, list, map, . . .
- cmath: special functions (inherited from C), e.g. `sqrt`
- cstdlib, cstring, cassert, . . . : C header files

RONTO

C++ Intro: Namespaces

- Variables and function, as well as variable types, have names.
- In larger projects, name clashes can occur.
- Solution: put all functions, types, ...in a namespace:

```
namespace nsname {
...
}
```

- Effectively prefixes all of ... with *nsname::*
  Example:
  `std::cout << "Hello, world" << std::endl;`
- Many standard functions/types are in namespace std.
- To omit the prefix, do using namespace *nsname*;
- Can selectively omit prefix, e.g., `using std::vector;`

IO Streams

In C++, stream object are responsible for I/O.
You can output an object `obj` to a stream `str` simply by

```
str << obj
```

while you can read an object `obj` from a stream `str` simply by

```
str >> obj
```

The stream will encode these object in ascii format, provided a proper operator is defined (true for the standard c++ types).

Standard streams

- `std::cout` For output to the screen (buffered)
- `std::cin` For input from the keyboard
- `std::cerr` For error messages (by default to the screen too)

These are defined in the header file `iostream`

IO Streams - an example

```
#include <iostream>
int main() {
    std::cout << "Print a number:  " << std::endl;
    int i;
    std::cin >> i;
    std::cout << "Twice that is:  " << 2*i << std::endl;
    return 0;
}
```

Streams - File IO

- Classes for file IO are defined in the header `fstream`.
- The `ofstream` class is for output to a file.
- The `ifstream` class is for input from a file.
- You have to declare an object of these classes first.
- Then you can use the streaming operators << and >>.
- Use member functions `read`/`write` to read/write binary.

Streams - File IO Examples

```
#include <fstream>
int main() {
    std::ofstream fout("out.txt");
    int x = 4;
    float y = 1.5;
    fout << x << " " << y << std::endl;
    fout.close();
    return 0;
}
```

```
#include <fstream>
#include <iostream>
int main() {
    std::ifstream fin("out.txt");
    int x2;
    float y2;
    fin >> x >> y;
    fin.close();
    std::cout << "x=" << x << " y=" << y <<std::endl;
    return 0;
}
```