

Scientific Computing for Physicists (PHY1610H)

Ramses van Zon

Winter 2022

Course Topics

This course aims at making you a more productive and efficient computational scientist.

It will cover best practices in scientific computing and programming skills, optimization and a bit of parallel programming.

There are three main themes in this course:

- 1 Scientific Software Development
- 2 Numerical Tools for Physical Scientists
- 3 High Performance Scientific Computing

Your Instructor

Who am I?

- My name is [Ramses van Zon](#)
- I am a High-Performance Computing Analysts at SciNet (SciNet is the supercomputing center at the University of Toronto)
- My Ph.D. was in Mathematical Physics, then I postdoc-ed in Chemical and Theoretical Physics, including a fair amount of molecular dynamics simulations and other computational projects.
- Nowadays, I am involving in training and education and all kinds of other aspects of running and supporting “high performance computing”.

The TA for this course is [Alistair Duff](#). He'll be helping with the grading of the assignments. He's taken this course in the past, so he knows what you're going through.

What is SciNet?



SciNet is UofT's supercomputer centre, which hosts and supports one of Canada's fastest supercomputers available to academic researchers.

We also do a lot of other teaching (Bash, Python, R, Fortran, C++, GPU programming, databases, machine learning, parallel programming, visualization, . . .)

Course website

<https://scinet.courses/1199>

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

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

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

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

The screenshot shows a web browser window displaying the course page for PHY1610 Scientific Computing for Physicists (Winter 2022) on the SciNet platform. The page title is "PHY1610 Scientific Computing for Physicists (Winter 2022)". The breadcrumb trail is "Home / Courses / Current Courses / PHY1610 - Winter 2022". The main content area features the course title and a description: "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 for developing software as it applies to scientific computations, common numerical techniques and packages, and aspects of high performance computing. While we will introduce the C++ language, in one language or another, students should already have some programming experience. Despite the title, this course is suitable for many physical scientists (chemists, astronomers, ...).". Below the description, it states: "This is a graduate course that can be taken for graduate credit by UoT PhD and MSc students. Students that wish to do so, should enrol using ACORN/ROSI." The teacher is listed as "Ramses van Zon" and the start date is "11 Jan 2022". A table of contents on the right side includes "General", "Course Description", "Syllabus", "Forum for questions regarding the course material", and "Announcements". At the bottom right, there is a calendar for January 2022.

Accounts, assignments, . . .

- You'll have access to SciNet's [teaching cluster](#) using a temporary student account, or your SciNet-enabled Compute Canada account.

```
ssh USERNAME@teach.scinet.utoronto.ca
```

- If you do not have a Compute Canada account, your login name on the education site is something that starts with [tmp_...](#)
- Your USERNAME for the Teach cluster is different from that, it will be of the form [lcl_uotphy1610s...](#)

You should've received this USERNAME and it's password by email.

- Initially, you can choose to do the homework assignments on your own computer, provided it has a unix-like environment with the `g++` compiler, `make`, and `git`.

Assignments are marked on how they can be compiled and run on the Teach cluster.

- Get a CC/SciNet account, if you want to keep working on SciNet after the course. See www.scinet.utoronto.ca/getting-a-scinet-account

Grading scheme

- **Near-weekly programming assignments** posted on the website.
- These assignments are **due the next week**.
- Each student should submit their own work.
- 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 5 points per day, out of the 100 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 instructor’s choice can be given at the end of the course.

Zoom, lectures, office hours, email, . . .

Zoom lectures

Lectures takes place via Zoom, at least for the next few weeks, following UofT guidelines.

You will need to click on the Zoom link from the course website, then enter the meeting password.

You'll be put in the waiting room until the host lets you in.

Lectures are recorded and posted on the site afterwards (often towards the end of the day).

Office hours

For the duration of the course, Zoom office hours will be on

- Wednesdays from 2:00 pm to 3:00 pm, and
- Fridays from 12 noon to 1 pm.

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

For questions regarding the course, use the forum on the course website or use the email courses@scinet.utoronto.ca.

Course lecture notes

For further reading, you might like the (incomplete) lecture notes of the course two years ago:

<https://support.scinet.utoronto.ca/materials/sclecturenotes.html>

© 2020 Ramses van Zon & Marcelo Ponce

Preface

I Introduction

- 1 What is Scientific Computing?
- 2 Examples of Scientific Computing
- 3 What You Will Need

II Scientific Software Engineering

- 1 Basics of Programming
- 2 C++
- 3 About "Best Practices"
- 4 Modularity
- 5 Defensive Programming
- 6 Testing
- 7 Version Control
- 8 Documentation
- 9 Using External Libraries
- 10 File IO & File Formats

III Numerical Techniques in Scientific Computin

- 1 Numerics
- 2 Root Finding
- 3 Optimization
- 4 Ordinary Differential Equations
- 5 Partial Differential Equations
- 6 Linear Algebra
- 7 Fourier Transform
- 8 Fitting
- 9 Random Numbers

IV High Performance Computing

- 1 Introduction to parallel computing
- 2 Measuring Performance
- 3 Optimizing Serial Performance
- 4 Job scheduling and load balancing

Lecture Notes on Scientific Computing with a focus on developing research software for the physical sciences

Ramses van Zon Marcelo Ponce

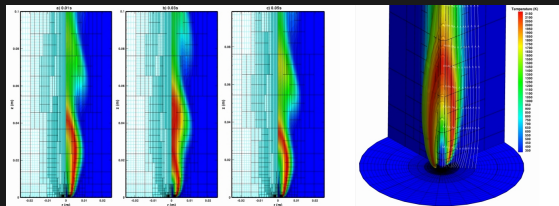
January 6, 2020

Preface

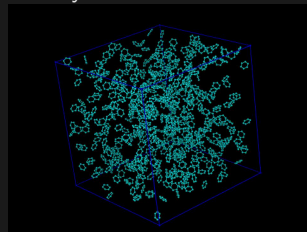
These notes are still under development. Contact us if you notice typos or errors of any kind or if you have suggestions for improvement.

Examples of Scientific Computations

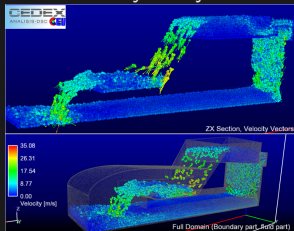
- Computational Fluid Dynamics



- Molecular Dynamics



- Smooth Particle Hydrodynamics



- Bioinformatics



- and many more...

Course Outline

1. Software development

- C++ intro
- Modular programming
- Building software with make
- Arrays and object
- Version control with git
- Unit testing
- I/O

2. Numerical tools

- Using libraries
- Ordinary differential equations
- Partial differential equations and lin. algebra
- Fast Fourier transforms
- Random numbers and Monte Carlo
- Molecular Dynamics

3. High-performance computing

- Profiling tools
- Intro to parallel computing
- Batch processing
- Shared memory programming
- Distributed parallel programming
- GPU programming

Section 1

Scientific Software Development

Recap of basic programming concepts

- 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 concepts: Programs and functions

- 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**.

Programming concepts: 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.

Programming concepts: 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)

Programming concepts: Input/Output

- Programs can receive input
- Input can take many, many different forms:
 - ▶ Interactive input from users (keyboard, mouse, ...)
 - ▶ Files with parameters
 - ▶ Files with data
 - ▶ Input from other programs
 - ▶ Input from the (inter)net
- Programs can (should) produce output
- Output can take many different forms too:
 - ▶ Output to console (text to screen)
 - ▶ Graphics output
 - ▶ Output to files
 - ▶ Output to other programs
 - ▶ Response to web requests

C++

We'll be using the C++ language in this course.

It's not the simplest language, but it is a language that can cover all cases we want in this course.

Advantages

- High performance
- Both low and high level programming
- Ubiquitous and standardized
- Useful libraries
- Modular design
- Supports many parallelization techniques

Disadvantages

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

Note: Fortran mostly has these advantages as well, but a choice had to be made.

Section 2

C++ Introduction

C++ Introduction

- C++ is **compiled** languages: their basic 'actions' are to be compiled into a set of basic 'native' instructions that the processor can execute.
- C, upon which C++ builds, was designed for (unix) system programming.
- C has a very small base.
- Most functionality is in (standard) libraries.
- For definiteness sake, let's say we use the **C++17** standard.

C++ Introduction: Basic C++ programming

The following code print “Hello, world!” on screen:

```
// Hello world program in C++  
  
#include <iostream>  
using std::cout;  
  
int main()  
{  
    cout << "Hello, world!\n";  
}
```

This might seem like too much code for just that. There’s some “boilerplate” lines here without which it does not work.

But the amount of boilerplate code does not grow linear with code, so in actual code, the signal/noise ratio is better.

To run this, we need to compile the code.

- 1 We’ll do this on the teach cluster:

```
$ ssh USERNAME@teach.scinet.utoronto.ca
```

- 2 First, avail yourself of a g++ compiler:

```
$ module load gcc/9
```

- 3 Type the code in a text editor, and save it:

```
$ nano helloworld.cpp
```

You can use `vi` or `emacs` as well.

- 4 Then we compile this into an executable

```
$ g++ -std=c++17 -o helloworld helloworld.cpp
```

- 5 And finally we run it.

```
$ ./helloworld  
Hello, world!
```

Super-short intro to the shell

Command prompt

There is a prompt, e.g. "rzon@teach01:~>" after which you can type in commands.

Any command you type at the prompt is read by a 'shell interpreter'. Teach uses the 'bash' shell.

Current directory

You are always "in" a current directory/folder in the file system tree. Your default directory, called your "home" directory, is where you start.

You can change to a directory with `cd DIRNAME`

- ~ is a shorthand for that home directory.
- . is a shorthand for the current directory
- .. is a shorthand for the parent directory.

Commands are either:

- built-in, or
- provided by executables in standard locations (encoded in the so called `PATH` variable), or
- executables of which the path is specified

Examples:

- List the files in the current directory with `ls`.
- If the current directory contains an executable "first", execute it with the command `./first`.
- Connect to a different computer with `ssh`.

After a command, you can optionally have more words, called the "arguments" of the command. What those arguments do, depends on the command.

Tips

Getting a terminal shell

For windows, get MobaXterm or use the Linux Subsystem for Windows.

It comes with ssh, so you can connect to teach using the ssh command.

For Mac and Linux, find your terminal application. It should also already come with the ssh command.

Editing code

Text-based editing of (code) files in the shell can be done using different applications.

- 'vi' is ubiquitous but not loved by all.
- 'emacs' is often available. Also not loved by all.
- 'nano' is a beginner friendly editor because all the possible actions are visible on the screen.
- Advanced: 'sshfs' can be used to make the files on the Teach cluster available on your local machine, then you can use your favourite local editor.
- VS code and other GUI editors? Anything is possible but they are usually slow and error prone to setup on remote systems.

Back to the C++ example

Here again is the code that print “Hello, world!” on screen:

```
// Hello world program in C++  
  
#include <iostream>  
using std::cout;  
  
int main()  
{  
    cout << "Hello, world!\n";  
}
```

- Lines starting with `//` are ignored by the compiler.
- Printing to screen is in a library `iostream` which needs to be included
- We tell the compiler that we’re using the object `cout` (console output)
- `int main` is function, and is called when the app is run.
- What that function does is enclosed in curly braces `{` and `}`.
- `cout << THING` prints that THING.
- Statements end in a semi-colon `;` .
- `"\n"` means the next console output should start on a newline

Another C++ Example: Input and variables

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string name;
    cout << "Type your name: ";
    cin >> name;
    cout << "Type your age: ";
    int age;
    cin >> age;
    cout << "You typed: \n"
         << "Name: " << name << "\n"
         << "Age:  " << age << "\n";
}
```

- This program uses a lot of the `std::` objects, so we import all of that namespace.
(not always a good idea)
- `int main`, starts by defining a variable named `name` of type `string`. All variables are typed in C++
- It reads from `cin` (console in, i.e., keyboard) into `name`
- It also reads an age, which is an integer.
- And it reports

Let's add a conditional statement

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string name;
    cout << "Type your name: ";
    cin >> name;
    cout << "Type your age: ";
    int age = -1;
    cin >> age;
    if (age <= 0) {
        cout << "Something is wrong!\n";
    } else {
        cout << "You typed: \n"
             << "Name: " << name << "\n"
             << "Age:  " << age << "\n";
    }
}
```

- Depending on the age variable, the program prints one thing or another, using if/else.
- Note that the code for the “one thing” has to be in a code block, delineated by curly braces.
- Similar for “another” thing.

Return values

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string name;
    cout << "Type your name: ";
    cin >> name;
    cout << "Type your age: ";
    int age = -1;
    cin >> age;
    if (age <= 0) {
        cout << "Something is wrong!\n";
        return 1;
    } else {
        cout << "You typed: \n"
            << "Name: " << name << "\n"
            << "Age:  " << age << "\n";
        return 0;
    }
}
```

```
$ g++ -std=c++17 -o main main.cpp
$ echo Alex -1 | ./main
Something is wrong
$ echo $?
1
$ echo Alex 48 | ./main
You typed:
Name: Alex
Age: 48
$ echo $?
0
```

- In addition to errors writing to screen, we return a number to the shell indicating success (0) or failure (non-zero).
- In bash, the value is stored in the variable `?`.
- Here, instead of typing the input, we have bash type it using `echo`, whose output got “piped” as input into `main`.

Repetition

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string name;
    cout << "Type your name: ";
    cin >> name;
    cout << "Type your age: ";
    int age = -1;
    cin >> age;
    while (age <= 0) {
        cout << "Something is wrong!\n";
        cout << "Type your age again: ";
        cin >> age;
    }
    cout << "You typed: \n";
    cout << "Name: " << name << "\n";
    cout << "Age: " << age << "\n";
}
```

- The idea here is to keep asking numbers for the age variable until a positive one is given.
- The `while` construct is good for this.
- But this can fail if we do not give an integer.

Repetition, fails better

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string name;
    cout << "Type your name: ";
    cin >> name;
    int age = -1;
    bool haveint = false;
    while (not haveint) {
        string ageword;
        cout << "Type your age: ";
        cin >> ageword;
        age = stoi(ageword);
        haveint = true;
    }
    cout << "You typed: \n";
    cout << "Name: " << name << "\n";
    cout << "Age: " << age << "\n";
}
```

Repetition, catching failures using exceptions

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string name;
    cout << "Type your name: ";
    cin >> name;
    int age = -1;
    bool waiting_for_valid_int = true;
    while (waiting_for_valid_int) {
        string ageword;
        cout << "Type your age: ";
        cin >> ageword;
        try {
            age = stoi(ageword);
            waiting_for_valid_int = false;
        } catch (invalid_argument& e) {
            cerr << "Error: invalid input\n";
        }
    }
    cout << "You typed: \n"
         << "Name: " << name << "\n"
         << "Age: " << age << "\n";
}
```

- Exceptions can be used to catch unexpected events, like entering a non-number for age.
- This goes via the try/catch construct.
- If stoi encounters an error, an **exception** is “thrown”.
- The exception is caught by the catch clause (in fact of a specific type).

Arrays

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string name;
    cout << "Type your name: ";
    cin >> name;
    const int nmax = 10;
    int age[nmax] = {0};
    int num = nmax;
    for (int i = 0; i<nmax && num == nmax; i++) {
        bool waiting_for_valid_int = true;
        while (waiting_for_valid_int) {
            string ageword;
            cout << "Type your pet's age (-1 to stop):";
            cin >> ageword;
            try {
                age[i] = stoi(ageword);
                waiting_for_valid_int = false;
            } catch (std::invalid_argument& e) {
                std::cerr << "Error: invalid input\n";
            }
        }
    }
}
```

```
    if (age[i] < 0)
        num = i;
}
cout << "You typed: \n"
    << "Name: " << name << "\n";
    << "Ages:";
for (int i = 0; i<num; i++) {
    cout << " " << age[i];
}
cout << "\n";
}
```

- Here we want to get several numbers and store them.
- C++ inherited "automatic arrays" from C. age is an example of such an array.
- Square brackets are used for indexing.
- The first element is element [0]
- The for loop is suitable for iterating over such an array.

Vectors

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
int main() {
    string name;
    cout << "Type your name: ";
    cin >> name;
    const int nmax = 10;
    vector<int> age;
    int num = nmax;
    for (int i = 0; i<nmax; i++) {
        bool waiting_for_valid_int = true;
        while (waiting_for_valid_int) {
            string ageword;
            cout << "Type your kid's age (-1 to stop):";
            cin >> ageword;
            try {
                age.push_back(stoi(ageword));
                waiting_for_valid_int = false;
            } catch (std::invalid_argument& e) {
                std::cerr << "Error: invalid input\n";
            }
        }
    }
}
```

```
        if (age[i] < 0)
            break;
    }
    cout << "You typed: \n";
    cout << "Name: " << name << "\n";
    cout << "Ages:";
    for (int a: age)
        cout << " " << a;
    cout << "\n";
}
```

- Here again we want to get several numbers and store them.
- But we're using the C++ standard vector.
- These have variable sizes.
- They also allow **range-based for loop**.

Functions

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
string getword(const string& prompt) {
    string result;
    cout << prompt;
    cin >> result;
    return result;
}
int getint(const string& prompt) {
    while (true) {
        string ageword = getword(prompt);
        try {
            return stoi(ageword);
        } catch (invalid_argument& e) {
            cerr << "Error: invalid input\n";
        }
    }
}
```

```
int main() {
    string name = getword("Type your name: ");
    const int nmax = 10;
    vector<int> age;
    while (true) {
        int thisage=getint("Type kid's age (-1 stops):");
        if (thisage != -1)
            age.push_back(thisage);
        if (age.size() == nmax or thisage == -1)
            break;
    }
    cout << "You typed: \n"
         << "Name: " << name << "\n"
         << "Ages:";
    for (int a: age)
        cout << " " << a;
    cout << "\n";
}
```

C++ Summary

These are the elements of C++ we'll be using in this class:

- Variables
- Functions
- Function arguments by value and by reference
- Loops
- Pointers
- Dynamically allocated arrays
- Conditionals
- Objects¹
- Templates¹
- Libraries
- Namespaces
- IO and file streams
- Compiling and linking

C++ Overview: 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

Examples:

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

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

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

```
bool itis2018 = false;
```

The type can be preceded by `const` to make it immutable.

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

C++ Overview: 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 uses ("calls") the functions, we only need to know its signature or interface;

To make the signature known, one has to place 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.

C++ Overview: Functions, an example

```
// funcexample.cpp

// external function prototypes:
#include <cmath>

// function prototype:
double arithmetic_mean(double a, double b);

// main function to call when program starts:
int main()
{
    double x = 16.3;
    double y = 102.4;
    return arithmetic_mean(x,y);
}

// function definition:
double arithmetic_mean(double a, double b)
{
    return sqrt(a*b);
}
```

```
$ ssh USERNAME@teach.scinet.utoronto.ca

$ module load gcc

$ g++ -std=c++17 -o funcexample funcexample.cpp

$ ./funcexample
$ echo $?
40
```

C++ Overview: Functions

- Function declaration (prototype/signature/interface)

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

argument-spec = comma separated list of variable definitions

- Function definition (code/implementation)

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

Functions which do not return anything have to be declared with a `returntype` of `void`.

Functions which have a non-void return type must have a `return` statement (except `main`).

- Function call

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

argument-list = comma separated list of values

C++ Overview: Pass by value or by reference

Passing function arguments by value

```
// passval.cpp
void inc(int i)
{
    i = i+1;
}

int main()
{
    int j = 10;
    inc(j);
    return j;
}
```

```
$ g++ -std=c++17 -o passval passval.cpp
$ ./passval
$ echo $?
10
$
```

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

C++ Overview: Pass by value or by reference

Passing function arguments by reference

```
// passref.cpp
void inc(int &i)
{
    i = i+1;
}

int main()
{
    int j = 10;
    inc(j);
    return j;
}
```

```
$ g++ -std=c++17 -o passref passref.cpp
$ ./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 an alias for j, j reflects this change.

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

Assignment

$a=b$ Assign a 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

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 are true (also: a and b)

$a||b$ either a or b is true (also: a or b)

Logic/Numeric

$a<b$ is a less than b

$a>b$ is a greater than b

$a<=b$ is a less then or equal to b

$a>=b$ is a greater than or equal to b

What is 1/4?

$$1/4 = 0$$

Why?

- In $1/4$ both operands, i.e., 1 and 4, are integers.
- Hence, the result of $1/4$ is the integer part of the division, which is 0.
- Generally, 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 the types of the operands.

Fix: Convert between types. In C/C++ this is called **casting**.

Casting one numeric type into another

Treat the type as a function.

Example:

```
// 1over4.cpp
#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++17 1over4.cpp -o 1over4
```

```
$ ./1over4
0 0.25 0
```

Note: the 'proper' C++ way to do casting is to use `static_cast<TYPE>(value)` or `reinterpret_cast<TYPE>(value)`.

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

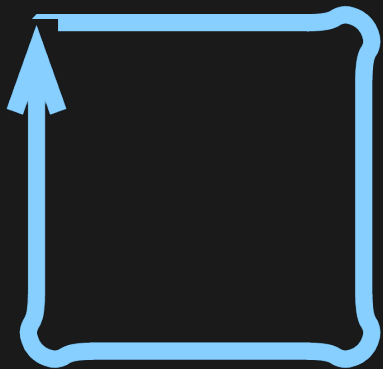
```
1.0/4.0
```

The expression may be a function call too, so that in this **example**:

```
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++ Overview: 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++ Overview: Loops

Two forms:

- for loop

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

- while loop

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

You can use the break statement to exit the loop.

For-loop example:

```
// count.cpp  
  
#include <iostream>  
  
int main()  
{  
    for (int i=1; i<=10; i++) {  
        std::cout << i << " ";  
    }  
    std::cout << std::endl;  
}
```

```
$ g++ -std=c++17 -o count count.cpp  
$ ./count  
1 2 3 4 5 6 7 8 9 10
```

Advanced: third loop form

There's actually a third form of `for`, called a range-based `for`:

```
for (type var: iterable-object-or-expression ) {
    statements
}
```

Example:

```
// rangebasedfor.cpp

#include <iostream>

int main()
{
    for (int i: { 1,2,3,4,5,6,7,8,9,10 } ) {
        std::cout << i << " ";
    }
    std::cout << std::endl;
}
```

```
$ g++ -std=c++17 -o rangebasedfor rangebasedfor.cpp
$ ./rangebasedfor
1 2 3 4 5 6 7 8 9 10
$
```

C++ Overview: Pointers

- Pointers are 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.
- Null pointer, denoted by `nullptr`, points to nowhere.
- Pointers are good for:
 - ▶ Arrays
 - ▶ Dynamic memory allocation
 - ▶ Linked lists, binary trees, ...
 - ▶ Calling C library functions

Definition:

```
type* name ;
```

Assignment (“address-of”):

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

Deferencing (“content-at”):

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

Example:

```
int main()
{
    int a = 5; // a equal to 5
    int* b = &a; // b points to a
    *b = 7; // *b is equivalent to a
    return a; // so this returns 7
}
```

*Note: These are so-called **raw pointers**, in contrast with smart pointers that are in the C++ standard library.*

C++ Overview: Automatic arrays

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

(square brackets are not indicating an optional part here, but are part of the syntax)

- name is equivalent to a pointer to the first element.
- Access to elements: `name[i]`.
- C/C++ arrays are zero-based.
- They're dangerous.

C++ Overview: Automatic arrays, example

```
// autoarr.cpp

#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;
}
```

```
$ g++ -std=c++17 -o autoarr autoarr.cpp
$ ./autoarr
25
$
```

What's so dangerous about automatic arrays?

- C standard only says at least one automatic array of at least 65535 bytes can be used.
- In practice, limit is set by compiler and OS.
- Compiler will not warn about the limit; the program will just crash.

C++ Overview: Automatic arrays, example

```
// autoarr1e8.cpp

#include <iostream>

int main()
{
    int a[100000000] = { 2,3,4,6,8,2 } ;
    int sum = 0;
    for (int i=0;i<100000000;i++) {
        sum += a[i];
    }
    std::cout << sum << std::endl;
}
```

```
$ g++ -std=c++17 -o autoarr autoarr.cpp
$ ./autoarr
25
$
```

```
$ g++ -std=c++17 -o autoarr1e8 autoarr1e8.cpp
$ ./autoarr1e8
Segmentation fault (core dumped)
$
```

What's so dangerous about automatic arrays?

- C standard only says at least one automatic array of at least 65535 bytes can be used.
- In practice, limit is set by compiler and OS.
- Compiler will not warn about the limit; the program will just crash.

C++ Overview: Dynamically allocated array

Dynamically allocated arrays are defined as a pointer to memory:

```
type* name ;
```

Allocated using the keyword new :

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

(the square brackets are part of the syntax)

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 given back.
- Must deallocate, or you'll have a memory leak.

Array allocation - Improved version

```
// dynarr.cpp

#include <iostream>

int main()
{
    int* a = new int[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;
    delete[] a;
}
```

```
$ g++ -std=c++17 -o dynarr dynarr.cpp
$ ./dynarr
25
$
```

Multidimensional arrays, you ask?

Unfortunately, no fully dynamic multi-dimensional version of the new keyword exists C++.

More about multi-dimensional arrays and other data structures in a later class.

Dynamic allocation of single variables

One can also dynamically allocate a single variable:

```
int main() {
    double* v = new double;
    *v = 4.2;
    std::cout << *v << std::endl;
    delete v;
}
```

Note the absence of [] in the delete statement.

You might use this in more dynamic data structures.

Note: this is where smart pointers like a `unique_ptr` or `shared_ptr` is useful.

```
#include <memory>
int main() {
    std::unique_ptr<double> v = std::make_unique<double>();
    *v = 4.2;
    std::cout << *v << std::endl;
    // no delete necessary
}
```

Arrays as function arguments

Array expressions and pointers are equivalent. Consider e.g. a function to print an array of integers:

```
void printarr(int size, int x[])
{
    for (int i=0; i<size; i++) {
        std::cout << x[i] << " ";
    }
    std::cout << std::endl;
}
```

We would call this function with an automatic array as follows:

```
int main() {
    int numbers[4] = {1,2,3,4};
    printarr(4, numbers);
}
```

Here, the size of the array has to be explicitly given to the function as its first argument.

This is because the array variable `numbers`, which used as an expression for the second argument, is converted to a pointer to the first element of the array.

From this pointer, there is no way to deduce how big the array was.

Command Line Arguments

Linux commands can be followed by arguments.

To get their value in a C++ program, we need change from `int main()` to

```
int main(int argc, char* argv[])
{
    ....
}
```

where:

- `argc` is the number of arguments, where the command itself counts as an argument as well
- `argv` is an array of character string, with the first string, `argv[0]` equal to the command

Note this is precisely how we just saw that an array needs to be passed to a function.

All arguments are strings. To convert them to integers or floats, use functions like `atoi` and `atof`, e.g. `int n = atoi(argv[1]);` stores the integer value of the first command line argument into the variable `n`.

Command Line Arguments Example

```
#include <iostream>
int main(int argc, char* argv[]) {
    for (int i=0; i<argc; i++) {
        std::cout << argv[i] << std::endl;
    }
}
```

```
$ g++ -std=c++17 -o printargs printargs.cpp
$ ./printargs Hello There!
./printargs
Hello
There!
$
```

C++ Overview: Conditionals

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

Example:

```
int main(int argc, char* argv[]) {
    int n = atoi(argv[1]);
    if ( n <= 20 ) {
        int* a = new int[n];
        for (int i=0; i<n; i++) {
            a[i] = i+1;
        }
        printarr(n,a);
        delete[] a;
    } else {
        std::cout << "Number given is too large\n";
    }
}
```

```
$ g++ -std=c++17 -o ifm ifm.cpp
$ ./ifm 20
0 1 4 9 16 25 36 49 64 81 100
121 144 169 196 225 256 289
324 361
```

Change $n = 20$ to $n = 2000000000$:

```
$ ./ifm 2000000000
Number given is too large
$
```

C++ Overview: Exceptions

```
try {
    statements
}
catch (type varname) {
    statements
}
```

```
int main() {
    int n = 20;
    int* a;
    try {
        a = new int[n];
    }
    catch (std::bad_alloc b) {
        std::cout << "Error in main" << std::endl;
        return 1;
    }
    for (int i=0; i<n; i++) {
        a[i] = i+1;
    }
    printarr(n,a);
    delete[] a;
}
```

```
$ g++ -std=c++17 -o ifm ifm.cpp
$ ./ifm
0 1 4 9 16 25 36 49 64 81 100
121 144 169 196 225 256 289
324 361
```

Change `n = 20` to `n = 2000000000`:

```
$ g++ -std=c++17 -o ifm ifm.cpp
$ ./ifm
Error in main
$
```

C++ Overview: Objects

Classes are a generalization of types.

Objects are a generalization of variables.

Syntax similar to variable declarations

```
classname objectname;  
classname objectname(arguments);  
classname objectname{arguments};
```

Differences between classes and regular types

- Object declarations can have arguments, supplied to construct the object.
- An object has members (fields) and member functions (methods), accessed using the “.” notation.

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

- You can create your own classes (though this isn't required for your course work).

C++ Overview: Classes and objects

Example of a member function/method

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

Example of a member/field

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

What are those angular brackets with types in between them?

C++ Overview: Templates

Templates

- Some algorithms and classes depend on a type. E.g. an list of doubles, a list of ints, ...
These objects can often be implemented with the same code, except for a change in type.
- Using generic programming, one can write this code once, with one or more type parameters.
- In C++, generic programming uses templates.
- Type parameters appear in between angular brackets `<>` instead of parenthesis.
- Many templated functions and classes are in the standard library, which we'll use.

C++ Overview: Class Templates

Usage

To create an object from a template class called `tc`:

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

Examples:

```
std::complex<float> z;      // single precision complex number
std::vector<int> i(20);     // array of 20 integers
rarray<float,2> x(20,20);   // 2d array of 20x20 floats (using the rarray library)
```

C++ Overview: 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,
- algorithm: sort, find, min, max, ...
- cmath: special functions (inherited from C), e.g. sqrt
- cstdlib, cstring, cassert, : C header files

Example: sort an array

```
#include <iostream>
#include <algorithm>

int main()
{
    int* a = new int[6] { 2,3,4,6,8,2 };
    std::sort(&a[0], &a[6]);
    for (int i=0;i<6;i++) {
        std::cout << a[i] << std::endl;
    }
    delete[] a;
}
```

- The algorithm library contains a template function to sort containers.
- You give it the pointers (or iterators) to the beginning and to the end.
- The 'end' here is one further than the last element (this should sound familiar if you know Python's list slicing).

C++ Overview: Namespaces

- Variables and function, as well as variable types, have names.
- In larger projects, you could have variable types of the same name.
- To avoid such name clashes, one can use namespaces
- One usually puts all functions, types, ... of a module in a namespace:

```
namespace modname {  
    ...  
}
```

(namespace is the keyword, modname is an identifier of your choosing)

- Effectively prefixes all of ... with modname::

Example:

```
std::cout << "Hello, world" << std::endl;
```

- Many standard functions/types are in namespace std.

C++ Overview: Scope

Variables do not live forever, they have well-defined scopes in which they exist. These are the rules:

If you define a variable inside a code block, it exists only until the your code hits the closing curly brace (}) that correspond to the opening curly brace ({) that started the block. This is its **local scope**.

The variable will only be known in that code block and its subblocks.

If you call a function from a code block, variable from that block will not be known in the body of the function.

It is possible to define variables outside of any code block; these are global variables. Avoid those.

When a variable goes out of scope, the memory associated with it is returned to the system, except for memory that was dynamically allocated.

When that variable is an object, a special member function of it called the destructor is called. This gives objects that dynamically allocate memory the opportunity to delete that memory.

C++ Overview: IO Streams

In C++, stream objects 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 objects 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`.

C++ Overview: IO Streams 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;
}
```

C++ Overview: File Streams

- 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.

C++ Overview: File Streams Examples

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

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

Some online resource that may help you out

- <https://www.learncpp.com/cpp-tutorial>
- <https://www.cplusplus.com/doc/tutorial>
- https://w3schools.com/cpp/cpp_exercises.asp