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 Chemistical 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.

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 Chemistical 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, ...)

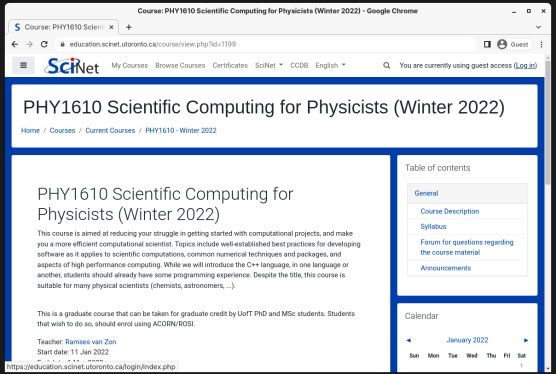
Course website

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.

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.

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.

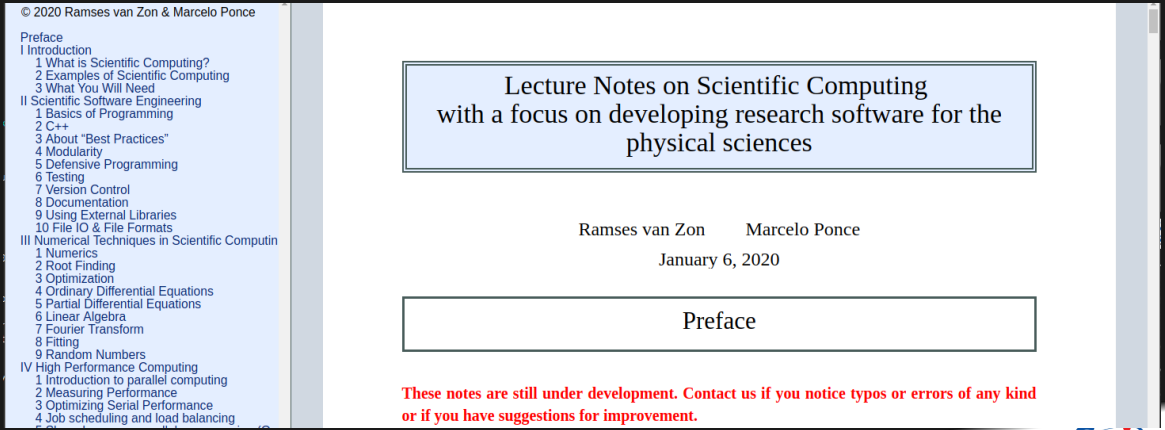
Questions/comments/concerns/etc. about the course?

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



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

Ramses van Zon Marcelo Ponce

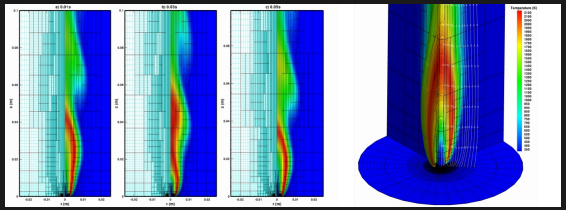
January 6, 2020

Preface

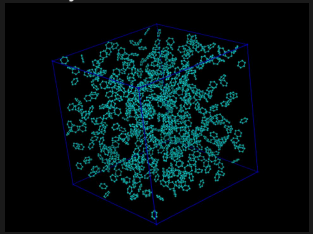
These notes are still under development. Contact us if you notice typos or errors of any kind or if you have suggestions for improvement.

Examples of Scientific Computations

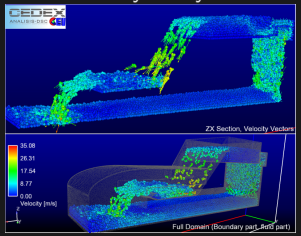
- Computational Fluid Dynamics



- 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 differental 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**.

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.

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

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

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 boilplate code does not grow linear with code, so in actual code, the signal/noise ratio is better.

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 boilplate 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 commmands.

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

Super-short intro to the shell

Command prompt

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

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.

Super-short intro to the shell

Command prompt

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

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

Super-short intro to the shell

Command prompt

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

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 curent 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 favorite local editor.

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 favorite local editor.
- VS code and other GUI editors? Anything is possible but there 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";
}
```

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 comments and ignored by the compiler.

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 comments and ignored by the compiler.
- Printing to screen is in a library `iostream` which needs to be `included`

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 comments and 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)

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 comments and 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.

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 comments and 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 }.

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 comments and 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`.

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 comments and 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` (`c`onsole `out`put)
- `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`.
- `"\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

```cpp
#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 can return a number to the shell to indicate success or failure.
- Returning 0 means success.
- In bash, the return value is given in the variable $?.
- By the way, instead of typing the input by hand, we have bash type it using echo.

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.

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 (std::invalid_argument& e) {
      std::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 "throw"
- 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;arra
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 (std::invalid_argument& e) {
      std::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";
}
```