Scientific Computing for Physicists

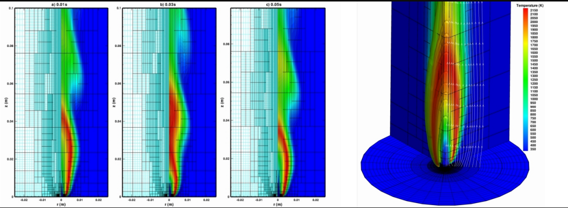
Ramses van Zon

PHY1610H 2023 Winter

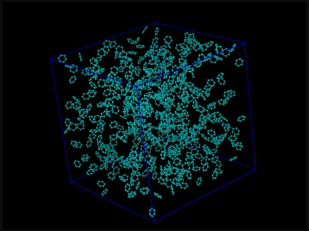
Course Intro

Examples of Scientific Computations

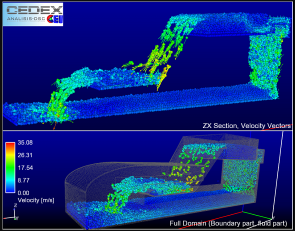
- Computational Fluid Dynamics



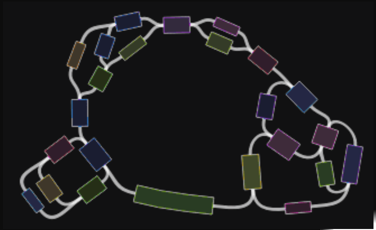
- Molecular Dynamics



- Smooth Particle Hydrodynamics



- BioInformatics



- and many more. . .

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

- My name is Ramses van Zon
- I am a High-Performance Computing Analyst at the SciNet HPC Consortium here at the University of Toronto.
- After a Ph.D. in Mathematical Physics, I postdoc-ed in Chemical and Theoretical Physics, which included development of molecular dynamics simulations and other computational projects.
- Currently, I'm involved in training and education and various aspects of running and supporting "high performance computing".
- The TA for this course is Kayhan Momeni. He'll be helping with the grading of the assignments. He has 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.

https://www.scinethpc.ca

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

https://scinet.courses

What is SciNet?

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

https://www.scinethpc.ca

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

https://scinet.courses

On a national level, we are a partner of the Digital Research Alliance of Canada (the successor of the Compute Canada Federation).

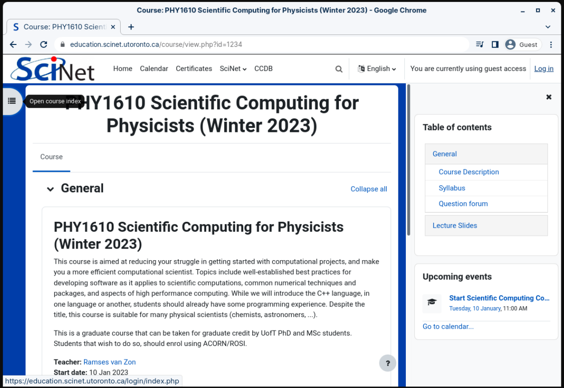
Course website

https://scinet.courses/1234

- Lectures
- Recordings
- Assignments
- Forum

Near-weekly assignments posted on the site on Thursdays.

To be able to hand in assignments and get course emails, you need to be able to login to the site (use your Alliance/CCDB account if you have one).



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

Accounts for this course

- If you do not have an Alliance account, your login name on the course site is something that starts with `tmp_...`
- For assignments, you'll have access to SciNet's Teach cluster using a separate account.

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

  Your USERNAME for the Teach cluster will be of the form `lcl_uotphy1610s...`

  You will received this USERNAME and its password by email.
- Initially, you can choose to do the assignments on your own computer, provided it has a unix-like environment with the g++ compiler, make, and git.
- If you want to keep working on SciNet after the course, get an Alliance/SciNet account, See www.scinet.utoronto.ca/getting-a-scinet-account

Assignments and grading

- **10 programming assignments** (so nearly weekly) will be posted on the website.
- These assignments are **due the next week.**
- Each student should hand in their own work.
- Assignments are graded on how they can be compiled and run on the Teach cluster.
- The average of the 10 assignments will make up your grade.
  (no midterm nor a final exam)
- All assignments need to be handed in for a passing grade.

Late penalty policy

- Assignments may be handed in up to 1 week after the due date, at a penalty of 5% per day.

  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.

Lectures, office hours, questions

Lectures

Lectures will be held in person on Tuesdays and Thursdays from 11:00 AM to 12:00 noon (EST) at the SciNet Teaching Room, which is located on the 11th floor of the West Tower of the MaRS building, 661 University Ave., Suite 1140, Toronto, ON M5G 1M1.

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

Lectures, office hours, questions

Lectures

Lectures will be held in person on Tuesdays and Thursdays from 11:00 AM to 12:00 noon (EST) at the SciNet Teaching Room, which is located on the 11th floor of the West Tower of the MaRS building, 661 University Ave., Suite 1140, Toronto, ON M5G 1M1.

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

Office hours

- In Person, Wednesdays from 2:00 pm to 3:00 pm, in the SciNet Teaching Room
- Virtually over Zoom, Fridays from 12 noon to 1 pm.

Lectures, office hours, questions

Lectures

Lectures will be held in person on Tuesdays and Thursdays from 11:00 AM to 12:00 noon (EST) at the SciNet Teaching Room, which is located on the 11th floor of the West Tower of the MaRS building, 661 University Ave., Suite 1140, Toronto, ON M5G 1M1.

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

Office hours

- In Person, Wednesdays from 2:00 pm to 3:00 pm, in the SciNet Teaching Room
- Virtually over Zoom, Fridays from 12 noon to 1 pm.

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

Use the forum on the course website or use the email courses@scinet.utoronto.ca.

Course Outline

1. Software development

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

Course Outline

1. Software development

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

2. Numerical tools

- Using libraries
- Ordinary differential equations
- Partial differental equations and linear algebra
- Fast Fourier transforms
- Random numbers/Monte Carlo

Course Outline

1. Software development

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

2. Numerical tools

- Using libraries
- Ordinary differential equations
- Partial differental equations and linear algebra
- Fast Fourier transforms
- Random numbers/Monte Carlo

3. High-performance

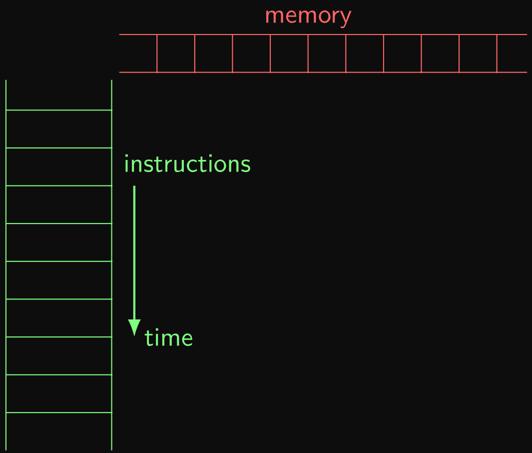
- Profiling tools
- Intro to parallel computing
- Batch processing
- Shared memory programming
- Distributed parallel programming
- GPU programming

Scientific Software Development

Basic programming concepts

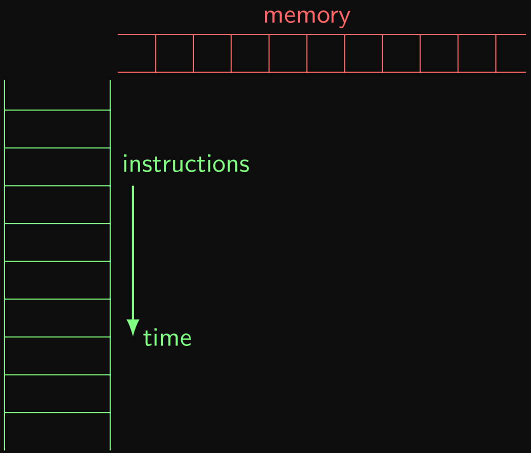
Basic programming concepts

Basic programming concepts



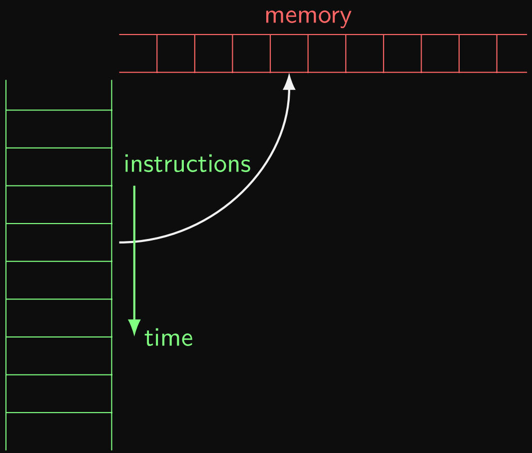
- **Von Neuman model:** A computer executes a set of instructions one-by-one on values stored in memory.

Basic programming concepts



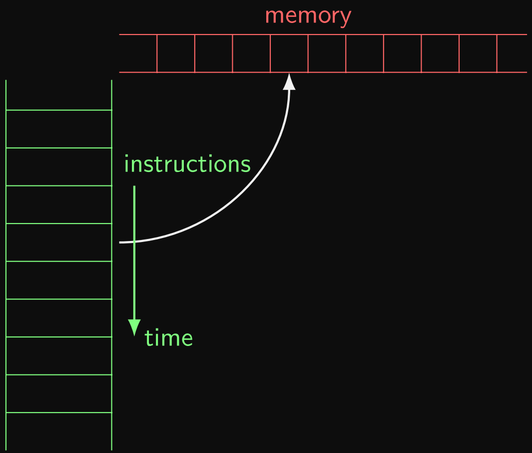
- **Von Neuman model:** A computer executes a set of instructions one-by-one on values stored in memory.
- A program contains a set of instructions.
- When a program is running, the data is holds in the computer's memory it called the program's state.

Basic programming concepts



- **Von Neuman model:** A computer executes a set of instructions one-by-one on values stored in memory.
- A program contains a set of instructions.
- When a program is running, the data is holds in the computer's memory it called the program's state.
- Each instruction will have a net effect on the program's state.

Basic programming concepts



- **Von Neuman model:** A computer executes a set of instructions one-by-one on values stored in memory.
- A program contains a set of instructions.
- When a program is running, the data is holds in the computer's memory it called the program's state.
- Each instruction will have a net effect on the program's state.
- There is limited set of predefined instructions, in terms of which we must express all other actions.

Programming concepts: Programs and functions

- An **algorithm** is a common pattern of actions to achieve a specific net effect (*computation*). Algorithms are described in words and math, and in a specific programming language.

Programming concepts: Programs and functions

- An **algorithm** is a common pattern of actions to achieve a specific net effect (*computation*). Algorithms are described in words and math, and in a specific programming language.

Ramanujan's formula for π:

$$\frac{1}{\pi} = \sum_{k=0}^{\infty} \frac{2\sqrt{2}}{99^2} \frac{(4k)!}{k!^4} \frac{26390k + 1103}{396^{4k}}$$

Programming concepts: Programs and functions

- An **algorithm** is a common pattern of actions to achieve a specific net effect (*computation*). Algorithms are described in words and math, and in a specific programming language.

Ramanujan's formula for π:

$$\frac{1}{\pi} = \sum_{k=0}^{\infty} \frac{2\sqrt{2}}{99^2} \frac{(4k)!}{k!^4} \frac{26390k + 1103}{396^{4k}}$$

Algorithm:

1. Start with k=0, and compute the k=0 term
2. Compute the term for the next k value
3. If less than the required accuracy, go to step 5
4. If not, add that term and repeat from step 2
5. Compute the inverse of the added terms.

Programming concepts: Programs and functions

- An **algorithm** is a common pattern of actions to achieve a specific net effect (*computation*). Algorithms are described in words and math, and in a specific programming language.

- A **function**, **procedure**, or **subroutine** is a set of actions, written in a specific programming language, that define a new action.

Ramanujan's formula for π:

$$\frac{1}{\pi} = \sum_{k=0}^{\infty} \frac{2\sqrt{2}}{99^2} \frac{(4k)!}{k!^4} \frac{26390k + 1103}{396^{4k}}$$

Algorithm:

1. Start with k=0, and compute the k=0 term
2. Compute the term for the next k value
3. If less than the required accuracy, go to step 5
4. If not, add that term and repeat from step 2
5. Compute the inverse of the added terms.

Programming concepts: Programs and functions

- An **algorithm** is a common pattern of actions to achieve a specific net effect (*computation*). Algorithms are described in words and math, and in a specific programming language.

- A **function**, **procedure**, or **subroutine** is a set of actions, written in a specific programming language, that define a new action.

Ramanujan's formula for π:

$$\frac{1}{\pi} = \sum_{k=0}^{\infty} \frac{2\sqrt{2}}{99^2} \frac{(4k)!}{k!^4} \frac{26390k + 1103}{396^{4k}}$$

```
#include <cmath>
double computeterm(int k) {
    return 2*sqrt(2)/pow(99,2)
          *tgamma(4*k+1)/pow(tgamma(k+1),4)
          *(26390*k+1103)/pow(396,4*k);
}
double computepi(double accuracy) {
    double sum = 0;
    for (int k=0;;k++) {
        double term = computeterm(k);
        if (term < accuracy) break;
        sum += term;
    }
    return 1/sum;
}
```

Programming concepts: Programs and functions

- An **algorithm** is a common pattern of actions to achieve a specific net effect (*computation*). Algorithms are described in words and math, and in a specific programming language.

- A **function**, **procedure**, or **subroutine** is a set of actions, written in a specific programming language, that define a new action.

- A **program** is a function that can be executed (a.k.a. application or app).

Ramanujan's formula for π:

$$\frac{1}{\pi} = \sum_{k=0}^{\infty} \frac{2\sqrt{2}}{99^2} \frac{(4k)!}{k!^4} \frac{26390k + 1103}{396^{4k}}$$

```
#include <cmath>
double computeterm(int k) {
    return 2*sqrt(2)/pow(99,2)
          *tgamma(4*k+1)/pow(tgamma(k+1),4)
          *(26390*k+1103)/pow(396,4*k);
}
double computepi(double accuracy) {
    double sum = 0;
    for (int k=0;;k++) {
        double term = computeterm(k);
        if (term < accuracy) break;
        sum += term;
    }
    return 1/sum;
}
```

Programming concepts: Programs and functions

- An **algorithm** is a common pattern of actions to achieve a specific net effect (*computation*). Algorithms are described in words and math, and in a specific programming language.

- A **function**, **procedure**, or **subroutine** is a set of actions, written in a specific programming language, that define a new action.

- A **program** is a function that can be executed (a.k.a. application or app).

```
#include <iostream>
int main() {
    std::cout<<"pi="<<computepi(1.0e-12)<<"\n";
}
```

Ramanujan's formula for π:

$$\frac{1}{\pi} = \sum_{k=0}^{\infty} \frac{2\sqrt{2}}{99^2} \frac{(4k)!}{k!^4} \frac{26390k + 1103}{396^{4k}}$$

```
#include <cmath>
double computeterm(int k) {
    return 2*sqrt(2)/pow(99,2)
          *tgamma(4*k+1)/pow(tgamma(k+1),4)
          *(26390*k+1103)/pow(396,4*k);
}
double computepi(double accuracy) {
    double sum = 0;
    for (int k=0;;k++) {
        double term = computeterm(k);
        if (term < accuracy) break;
        sum += term;
    }
    return 1/sum;
}
```

Programming concepts: Programs and functions

- An **algorithm** is a common pattern of actions to achieve a specific net effect (*computation*). Algorithms are described in words and math, and in a specific programming language.

- A **function**, **procedure**, or **subroutine** is a set of actions, written in a specific programming language, that define a new action.

- A **program** is a function that can be executed (a.k.a. application or app).

```
def main():
    print("pi=",computepi(1.0e-12))
if __name__ == "__main__":
    main()
```

Ramanujan's formula for π:

$$\frac{1}{\pi} = \sum_{k=0}^{\infty} \frac{2\sqrt{2}}{99^2} \frac{(4k)!}{k!^4} \frac{26390k + 1103}{396^{4k}}$$

```
from math import sqrt,factorial
def computeterm(k):
    return 2*sqrt(2)/99**2*(
          factorial(4*k)/factorial(k)**4
          *(26390*k+1103)/396**(4*k))
def computepi(accuracy):
    sum = 0.0
    k = 0
    while True:
        term = computeterm(k)
        if term < accuracy:
            break
        sum += term
        k += 1
    return 1/sum
```

Programming concepts: Languages

- The computer's Central Processing Unit (CPU) does not understand programming languages, only **machine code**.
- To execute code written in a programming language, one needs another program, either a
  - **Compiler**: translates source code files into **executable** or **object** files containing machine code.
  - **Interpreter**: does that translation on the fly, one line of code at a time.

C++ falls in the category of compiled programming languages.
Python is an example of an interpreted language.

Programming concepts: State

Programming concepts: State

- Program state is stored in memory.

Programming concepts: State

- Program state is stored in memory.
- At least part of the state is made up of the program's **variables**.

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 stored values that are assigned to a **variable name**.

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 stored 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: State

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

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 stored values that are assigned to a **variable name**.
- This variable name is associated with a portion of **memory** that holds the variable's value.
- What the variable stores can change in time.

Note on persistance

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 stored values that are assigned to a **variable name**.
- This variable name is associated with a portion of **memory** that holds the variable's value.
- What the variable stores can change in time.

Note on persistance

- The common definition of state above, involves only what is in memory.

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 stored values that are assigned to a **variable name**.
- This variable name is associated with a portion of **memory** that holds the variable's value.
- What the variable stores can change in time.

Note on persistance

- The common definition of state above, involves only what is in memory.
- When a program ends, its state is gone.

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 stored values that are assigned to a **variable name**.
- This variable name is associated with a portion of **memory** that holds the variable's value.
- What the variable stores can change in time.

Note on persistance

- The common definition of state above, involves only what is in memory.
- When a program ends, its state is gone.
- Files are a way to store data persistently, but fall under **I/O** (input/output)

Programming concepts: Control structures

- Some actions could be done conditionally on the state of the program and external input.

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.

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.
- These are usually some variation of an `if-then-else` statement.

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.
- These are usually some variation of an `if-then-else` statement.
- Repetition of a set of actions, *i.e.*, **loops**, are also a type of control structure: they keep doing the same while there are loop iterations left.

Programming concepts: I/O

Programs can receive input

- Interactive (keyboard, mouse, camera, mic)

Programming concepts: I/O

Programs can receive input

- Interactive (keyboard, mouse, camera, mic)
- Files containing parameters

Programming concepts: I/O

Programs can receive input

- Interactive (keyboard, mouse, camera, mic)
- Files containing parameters
- Files containing data

Programming concepts: I/O

Programs can receive input

- Interactive (keyboard, mouse, camera, mic)
- Files containing parameters
- Files containing data
- Input from other programs

Programming concepts: I/O

Programs can receive input

- Interactive (keyboard, mouse, camera, mic)
- Files containing parameters
- Files containing data
- Input from other programs
- Input from a local network or from the internet

Programming concepts: I/O

Programs can receive input

- Interactive (keyboard, mouse, camera, mic)
- Files containing parameters
- Files containing data
- Input from other programs
- Input from a local network or from the internet

Programs can (should) produce output.

- Output to console

Programming concepts: I/O

Programs can receive input

- Interactive (keyboard, mouse, camera, mic)
- Files containing parameters
- Files containing data
- Input from other programs
- Input from a local network or from the internet

Programs can (should) produce output.

- Output to console
- Graphical output

Programming concepts: I/O

Programs can receive input

- Interactive (keyboard, mouse, camera, mic)
- Files containing parameters
- Files containing data
- Input from other programs
- Input from a local network or from the internet

Programs can (should) produce output.

- Output to console
- Graphical output
- Output to files

Programming concepts: I/O

Programs can receive input

- Interactive (keyboard, mouse, camera, mic)
- Files containing parameters
- Files containing data
- Input from other programs
- Input from a local network or from the internet

Programs can (should) produce output.

- Output to console
- Graphical output
- Output to files
- Output to other programs

Programming concepts: I/O

Programs can receive input

- Interactive (keyboard, mouse, camera, mic)
- Files containing parameters
- Files containing data
- Input from other programs
- Input from a local network or from the internet

Programs can (should) produce output.

- Output to console
- Graphical output
- Output to files
- Output to other programs
- Response to web requests

Other programming paradigms

This overview described so-called imperative programming paradigm, in which a list of commands acting on data is executed in order.

Other programming paradigms

This overview described so-called imperative programming paradigm, in which a list of commands acting on data is executed in order.

There are also other paradigms:

Other programming paradigms

This overview described so-called imperative programming paradigm, in which a list of commands acting on data is executed in order.

There are also other paradigms:

- functional programming
- declarative programming
- object-oriented programming
- generic or metaprogramming.

Other programming paradigms

This overview described so-called imperative programming paradigm, in which a list of commands acting on data is executed in order.

There are also other paradigms:

- functional programming
- declarative programming
- object-oriented programming
- generic or metaprogramming.

Imperative programming mimics more or less what the computer actually does when running a program, and will be our main focus.

C++

Why 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.

Why 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-level and high-level programming
- Ubiquitous and standardized
- Useful libraries
- Modular design
- Supports imperative, functional, object-oriented, and metaprogramming
- Supports many parallelization techniques
- Interoperable with C and Fortran.

Why 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-level and high-level programming
- Ubiquitous and standardized
- Useful libraries
- Modular design
- Supports imperative, functional, object-oriented, and metaprogramming
- Supports many parallelization techniques
- Interoperable with C and Fortran.

Disadvantages

- Precise syntax
- Errors can be hard to interpret
- Non-interactive
- Steeper learning curve
- No standard portable graphics
- Susceptible to hidden performance pitfalls
- Susceptible to memory errors

Why 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-level and high-level programming
- Ubiquitous and standardized
- Useful libraries
- Modular design
- Supports imperative, functional, object-oriented, and metaprogramming
- Supports many parallelization techniques
- Interoperable with C and Fortran.

Disadvantages

- Precise syntax
- Errors can be hard to interpret
- Non-interactive
- Steeper learning curve
- No standard portable graphics
- Susceptible to hidden performance pitfalls
- Susceptible to memory errors

Note: Fortran shares many of the advantages.

C++ Introduction

- C++ is a **compiled** language: files containing the basic 'actions' are to be compiled into a set of basic machine language instructions that the CPU can execute.

C++ Introduction

- C++ is a **compiled** language: files containing the basic 'actions' are to be compiled into a set of basic machine language instructions that the CPU can execute.
- The C language, upon which C++ builds, was designed for (unix) system programming.

C++ Introduction

- C++ is a **compiled** language: files containing the basic 'actions' are to be compiled into a set of basic machine language instructions that the CPU can execute.
- The C language, upon which C++ builds, was designed for (unix) system programming.
- The C language has a very small base.

C++ Introduction

- C++ is a **compiled** language: files containing the basic 'actions' are to be compiled into a set of basic machine language instructions that the CPU can execute.
- The C language, upon which C++ builds, was designed for (unix) system programming.
- The C language has a very small base.
- Most functionality is in (standard) libraries.

C++ Introduction

- C++ is a **compiled** language: files containing the basic 'actions' are to be compiled into a set of basic machine language instructions that the CPU can execute.
- The C language, upon which C++ builds, was designed for (unix) system programming.
- The C language has a very small base.
- Most functionality is in (standard) libraries.
- Every 3 years, a new C++ standard comes out, which is by-and-large backwards compatible.

C++ Introduction

- C++ is a **compiled** language: files containing the basic 'actions' are to be compiled into a set of basic machine language instructions that the CPU can execute.
- The C language, upon which C++ builds, was designed for (unix) system programming.
- The C language has a very small base.
- Most functionality is in (standard) libraries.
- Every 3 years, a new C++ standard comes out, which is by-and-large backwards compatible.
- For definiteness sake, use the **C++17** standard.

C++ Introduction: Basic C++ programming

The following code prints “Hello, world!” on the console:

```
// @file helloworld.cpp
// Hello world program in C++
#include <iostream>
using std::cout;

int main()
{
   cout << "Hello, world!\n";
}
```

C++ Introduction: Basic C++ programming

The following code prints "Hello, world!" on the console:

```cpp
// @file helloworld.cpp
// Hello world program in C++
#include <iostream>
using std::cout;

int main()
{
   cout << "Hello, world!\n";
}
```

To run this, we need to compile the code.

C++ Introduction: Basic C++ programming

The following code prints "Hello, world!" on the console:

```
// @file helloworld.cpp
// Hello world program in C++
#include <iostream>
using std::cout;

int main()
{
   cout << "Hello, world!\n";
}
```

To run this, we need to compile the code.

1. We'll do this on the teach cluster:

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

C++ Introduction: Basic C++ programming

The following code prints “Hello, world!” on the console:

```
// @file helloworld.cpp
// Hello world program in C++
#include <iostream>
using std::cout;

int main()
{
   cout << "Hello, world!\n";
}
```

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/12
```

C++ Introduction: Basic C++ programming

The following code prints “Hello, world!” on the console:

```
// @file helloworld.cpp
// Hello world program in C++
#include <iostream>
using std::cout;

int main()
{
   cout << "Hello, world!\n";
}
```

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/12
```

3. Start a new code file in a text editor, *e.g.*

```
$ nano helloworld.cpp
```

C++ Introduction: Basic C++ programming

The following code prints “Hello, world!” on the console:

```
// @file helloworld.cpp
// Hello world program in C++
#include <iostream>
using std::cout;

int main()
{
   cout << "Hello, world!\n";
}
```

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/12
```

3. Start a new code file in a text editor, *e.g.*

```
$ nano helloworld.cpp
```

4. Type in the code, save it, and exit the editor.

C++ Introduction: Basic C++ programming

The following code prints “Hello, world!” on the console:

```cpp
// @file helloworld.cpp
// Hello world program in C++
#include <iostream>
using std::cout;

int main()
{
   cout << "Hello, world!\n";
}
```

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/12
```

3. Start a new code file in a text editor, *e.g.*

```
$ nano helloworld.cpp
```

4. Type in the code, save it, and exit the editor.
5. Then, compile this into an executable

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

C++ Introduction: Basic C++ programming

The following code prints "Hello, world!" on the console:

```
// @file helloworld.cpp
// Hello world program in C++
#include <iostream>
using std::cout;

int main()
{
   cout << "Hello, world!\n";
}
```

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/12
```

3. Start a new code file in a text editor, *e.g.*

```
$ nano helloworld.cpp
```

4. Type in the code, save it, and exit the editor.
5. Then, compile this into an executable

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

6. Finally, run it.

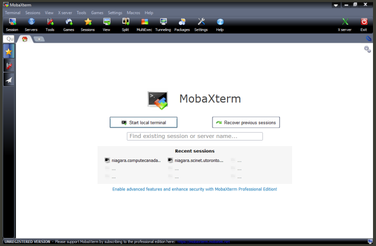
```
$ ./helloworld
Hello, world!
```

Short intro to the terminal a.k.a. console

How to get a terminal

On Windows

Get MobaXterm:

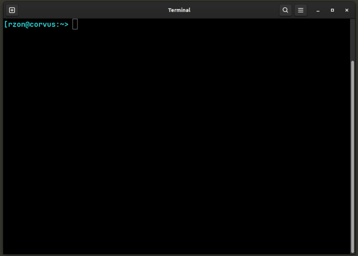


MobaXterm's local terminal runs the bash shell and comes with ssh and X11.

You can also use the Linux Subsystem for Windows.

On Linux

Find your terminal application.



The most common shell interpreter on Linux is bash.

It should have the ssh command.

On MacOS

Find your terminal application.

The default shell is zsh or bash, depending on the MacOS version.

It should have the ssh command.

You need Xquartz for remote X graphics to work.

Command line interface

Command line interface

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.

Command line interface

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.

Command line interface

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.

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

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.

Command line interface

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.

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

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.

Command 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`.

Command line interface

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

Command 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`.

Command line arguments

After a command, more words can be entered, the "arguments" of the command.

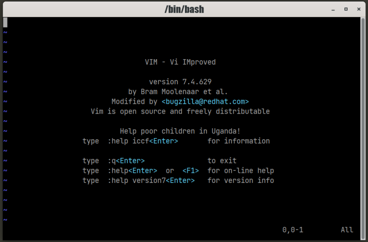
Tips on editing code

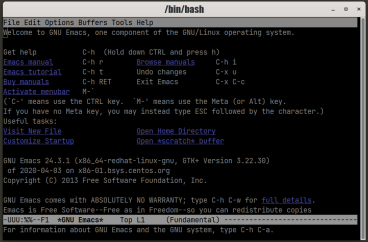
When logged into Teach with ssh, you cannot see the files on your computer.
Text-based editing of files in the terminal on Teach can be done using different applications.

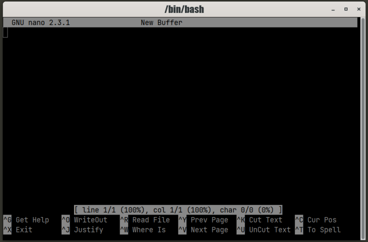
Tips on editing code

When logged into Teach with ssh, you cannot see the files on your computer.
Text-based editing of files in the terminal on Teach can be done using different applications.

vi	emacs	nano
ubiquitous but not loved by all.	often available; not loved by all.	beginner friendly editor







Note: VS code and other GUI editors can be slow and tricky to setup on remote systems.

C++ by Example

Back to the C++ example

Here is, again, the code that prints “Hello, world!”:

```
// @file helloworld.cpp
// Hello world program in C++
#include <iostream>
using std::cout;

int main()
{
   cout << "Hello, world!\n";
}
```

Back to the C++ example

Here is, again, the code that prints "Hello, world!":

```
// @file helloworld.cpp
// Hello world program in C++
#include <iostream>
using std::cout;

int main()
{
   cout << "Hello, world!\n";
}
```

Let's look at what each line in this code means:

Back to the C++ example

Here is, again, the code that prints “Hello, world!”:

```
// @file helloworld.cpp
// Hello world program in C++
#include <iostream>
using std::cout;

int main()
{
    cout << "Hello, world!\n";
}
```

Let’s look at what each line in this code means:

- Lines starting with // are comments and are ignored by the compiler.

Back to the C++ example

Here is, again, the code that prints "Hello, world!":

```
// @file helloworld.cpp
// Hello world program in C++
#include <iostream>
using std::cout;

int main()
{
    cout << "Hello, world!\n";
}
```

Let's look at what each line in this code means:

- Lines starting with // are comments and are ignored by the compiler.
- Printing to console is in a library called `iostream`, which needs to be `include`d

Back to the C++ example

Here is, again, the code that prints "Hello, world!":

```
// @file helloworld.cpp
// Hello world program in C++
#include <iostream>
using std::cout;

int main()
{
   cout << "Hello, world!\n";
}
```

Let's look at what each line in this code means:

- Lines starting with // are comments and are ignored by the compiler.
- Printing to console is in a library called 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 is, again, the code that prints "Hello, world!":

```
// @file helloworld.cpp
// Hello world program in C++
#include <iostream>
using std::cout;

int main()
{
   cout << "Hello, world!\n";
}
```

Let's look at what each line in this code means:

- Lines starting with // are comments and are ignored by the compiler.
- Printing to console is in a library called iostream, which needs to be included
- We tell the compiler that we're using the object cout (console output)
- int main is a function, and is, by definition, called when the program is run.

Back to the C++ example

Here is, again, the code that prints "Hello, world!":

```
// @file helloworld.cpp
// Hello world program in C++
#include <iostream>
using std::cout;

int main()
{
   cout << "Hello, world!\n";
}
```

Let's look at what each line in this code means:

- Lines starting with // are comments and are ignored by the compiler.
- Printing to console is in a library called iostream, which needs to be included
- We tell the compiler that we're using the object cout (console output)
- int main is a function, and is, by definition, called when the program is run.
- What that function does is enclosed in curly braces { and }.

Back to the C++ example

Here is, again, the code that prints “Hello, world!”:

```
// @file helloworld.cpp
// Hello world program in C++
#include <iostream>
using std::cout;

int main()
{
   cout << "Hello, world!\n";
}
```

Let’s look at what each line in this code means:

- Lines starting with // are comments and are ignored by the compiler.
- Printing to console is in a library called iostream, which needs to be included
- We tell the compiler that we’re using the object cout (console output)
- int main is a function, and is, by definition, called when the program is run.
- What that function does is enclosed in curly braces { and }.
- cout<<THING prints that THING.

Back to the C++ example

Here is, again, the code that prints "Hello, world!":

```
// @file helloworld.cpp
// Hello world program in C++
#include <iostream>
using std::cout;

int main()
{
   cout << "Hello, world!\n";
}
```

Let's look at what each line in this code means:

- Lines starting with // are comments and are ignored by the compiler.
- Printing to console is in a library called iostream, which needs to be included

- We tell the compiler that we're using the object cout (console output)
- int main is a function, and is, by definition, called when the program is run.
- What that function does is enclosed in curly braces { and }.
- cout<<THING prints that THING.
- Statements end in a semi-colon, *i.e.* ;

Back to the C++ example

Here is, again, the code that prints "Hello, world!":

```
// @file helloworld.cpp
// Hello world program in C++
#include <iostream>
using std::cout;

int main()
{
   cout << "Hello, world!\n";
}
```

Let's look at what each line in this code means:

- Lines starting with `//` are comments and are ignored by the compiler.
- Printing to console is in a library called `iostream`, which needs to be `include`d
- We tell the compiler that we're using the object cout (console output)
- `int main` is a function, and is, by definition, called when the program is run.
- What that function does is enclosed in curly braces `{` and `}`.
- `cout<<THING` prints that `THING`.
- Statements end in a semi-colon, *i.e.* `;`
- Strings, i.e., literal text that is not code, has to be given between quotation marks `"..."`.

Back to the C++ example

Here is, again, the code that prints “Hello, world!”:

```
// @file helloworld.cpp
// Hello world program in C++
#include <iostream>
using std::cout;

int main()
{
   cout << "Hello, world!\n";
}
```

Let’s look at what each line in this code means:

- Lines starting with // are comments and are ignored by the compiler.
- Printing to console is in a library called iostream, which needs to be included
- We tell the compiler that we’re using the object cout (console output)
- int main is a function, and is, by definition, called when the program is run.
- What that function does is enclosed in curly braces { and }.
- cout<<THING prints that THING.
- Statements end in a semi-colon, *i.e.* ;
- Strings, i.e., literal text that is not code, has to be given between quotation marks "...".
- \n inside a string is a newline and means the next console output should start on the next line.

Back to the C++ example

Here is, again, the code that prints "Hello, world!":

```
// @file helloworld.cpp
// Hello world program in C++
#include <iostream>
using std::cout;

int main()
{
   cout << "Hello, world!\n";
}
```

Let's look at what each line in this code means:

- Lines starting with `//` are comments and are ignored by the compiler.
- Printing to console is in a library called `iostream`, which needs to be `include`d
- We tell the compiler that we're using the object cout (console output)
- `int main` is a function, and is, by definition, called when the program is run.
- What that function does is enclosed in curly braces `{` and `}`.
- `cout<<THING` prints that `THING`.
- Statements end in a semi-colon, *i.e.* `;`
- Strings, i.e., literal text that is not code, has to be given between quotation marks `"..."`.
- `\n` inside a string is a newline and means the next console output should start on the next line.

Another C++ Example: Input and variables

Another C++ Example: Input and variables

```cpp
// @file inputex.cpp
#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";
}
```

Another C++ Example: Input and variables

```cpp
// @file inputex.cpp
#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 many `std::` objects, so we import all of that namespace.

  (not generally a good idea)

Another C++ Example: Input and variables

```
// @file inputex.cpp
#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 many `std::` objects, so we import all of that namespace.

  (not generally a good idea)

- `int main` starts by defining a variable named **name** of **type** `string`.
  All variables have a **type** in C++

Another C++ Example: Input and variables

```
// @file inputex.cpp
#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 many `std::` objects, so we import all of that namespace.

  (not generally a good idea)

- `int main` starts by defining a variable named **name** of **type** `string`.
  All variables have a **type** in C++

- It reads from `cin` (console `in`, *i.e.*, keyboard) into the existing variable `name`

Another C++ Example: Input and variables

```
// @file inputex.cpp
#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 many `std::` objects, so we import all of that namespace.

  (not generally a good idea)
- `int main` starts by defining a variable named **name** of **type** `string`.
  All variables have a **type** in C++
- It reads from `cin` (console in, *i.e.*, keyboard) into the existing variable `name`
- It also defines and reads an `age` variable, which is of type `int`.

Another C++ Example: Input and variables

```
// @file inputex.cpp
#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 many `std::` objects, so we import all of that namespace.

  (not generally a good idea)

- `int main` starts by defining a variable named **`name`** of **type** `string`.
  All variables have a **type** in C++

- It reads from `cin` (console `in`, *i.e.*, keyboard) into the existing variable `name`

- It also defines and reads an `age` variable, which is of type `int`.

- And it reports what was typed by the user.

Another C++ Example: Input and variables

```
// @file inputex.cpp
#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 many `std::` objects, so we import all of that namespace.

  (not generally a good idea)

- `int main` starts by defining a variable named **name** of **type** `string`.
  All variables have a **type** in C++
- It reads from `cin` (console `in`, *i.e.*, keyboard) into the existing variable `name`
- It also defines and reads an `age` variable, which is of type `int`.
- And it reports what was typed by the user.

Note that variables and their types must be defined before they can be used!

Let's add a conditional statement

```cpp
// @file inputex.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";
  } else {
     cout << "You typed: \n"
          << "Name: " << name << "\n"
          << "Age:  " << age << "\n";
  }
}
```

Let's add a conditional statement

```
// @file inputex.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";
  } 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`.

Let's add a conditional statement

```cpp
// @file inputex.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";
  } 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, *i.e.* `{...}` .

Let's add a conditional statement

```
// @file inputex.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";
  } 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, *i.e.* `{...}` .
- Similarly, the else code block is delineated by braces.

Let's add a conditional statement

```cpp
// @file inputex.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";
  } 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, *i.e.* {...} .
- Similarly, the else code block is delineated by braces.

Let's add a return value

```
// @file inputex.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;
  }
}
```

Let's add a return value

```cpp
// @file inputex.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;
  }
}
```

- In addition to errors writing to console, we return an exit code to the shell indicating success (0) or failure (non-zero).

Let's add a return value

```
// @file inputex.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;
  }
}
```

- In addition to errors writing to console, we return an exit code to the shell indicating success (0) or failure (non-zero).
- The value returned by `main` must be an `int`.

Let's add a return value

```cpp
// @file inputex.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;
  }
}
```

- In addition to errors writing to console, we return an exit code to the shell indicating success (0) or failure (non-zero).
- The value returned by `main` must be an `int`.

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

Let's add a return value

```
// @file inputex.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;
  }
}
```

- In addition to errors writing to console, we return an exit code to the shell indicating success (0) or failure (non-zero).
- The value returned by `main` must be an `int`.

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

- In *bash*, the exit code of the last executed command is stored in the variable `$?`.

Let's add a return value

```cpp
// @file inputex.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;
  }
}
```

- In addition to errors writing to console, we return an exit code to the shell indicating success (0) or failure (non-zero).
- The value returned by `main` must be an `int`.

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

- In *bash*, the exit code of the last executed command is stored in the variable `$?`.
- Here, *bash* types input with "`echo`" and "pipes" that into "`inputex`".

Let's add a return value

```
// @file inputex.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;
  }
}
```

- In addition to errors writing to console, we return an exit code to the shell indicating success (0) or failure (non-zero).
- The value returned by `main` must be an `int`.

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

- In *bash*, the exit code of the last executed command is stored in the variable `$?`.
- Here, *bash* types input with "`echo`" and "pipes" that into "`inputex`".

How to ask again: 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.

How to ask again: 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.

Arrays

```
#include <iostream>
#include <string>

using namespace std;
int main() {
  string name;
  cout << "Type your name: ";
  cin >> name;
  int nmax = 10;
  int age[nmax] = {0};
  int num;
  for (num = 0; num<nmax; num++) {
    string ageword;
    cout << "Type your pet's age (-1 to stop): ";
    cin >> ageword;
    age[num] = stoi(ageword);
    if (age[num] < 0)
      break;
  }
  cout << "You typed: \n";
  cout << "Name: " << name << "\n";
```

```
  cout << "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;
  int nmax = 10;
  vector<int> age;
  int num;
  for (num = 0; num<nmax; num++) {
    string ageword;
    cout << "Type your pet's age (-1 to stop): ";
    cin >> ageword;
    age.push_back(stoi(ageword));
    if (age[num] < 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.
- Can use square brackets are used for indexing, with the first element begin [0].
- But they also support range-based for loop.

Functions

The code is starting to look a bit messy; we can make it clearer with some 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) {
      cout << "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 your kid's age (or -1
    if (thisage != -1)
      age.push_back(thisage);
    if (age.size() == nmax or thisage == -1)
      break;
  }
  cout << "You typed: \n";
  cout << "Name: " << name << "\n";
  cout << "Ages:";
  for (int a: age) {
    cout << " " << a;
  }
  cout << "\n";
}
```

There are now separate functions dealing with reading in a word and an integer.

Dealing with errors

You may have noticed thet the `getint` function does something interesting to catch errors.

Rather than just saying

```
int getint(const string& prompt) {
  string ageword = getword(prompt);
  return stoi(ageword);
}
```

(which crashes the program if `ageword` does not contain an integer)

Dealing with errors

You may have noticed thet the `getint` function does something interesting to catch errors.

Rather than just saying

```
int getint(const string& prompt) {
  string ageword = getword(prompt);
  return stoi(ageword);
}
```

(which crashes the program if `ageword` does not contain an integer)

it does this:

```
int getint(const string& prompt) {
  while (true) {
    string ageword = getword(prompt);
    try {
      return stoi(ageword);
    } catch (invalid_argument& e) {
      cout << "Error: invalid input\n";
    }
  }
}
```

Dealing with errors

You may have noticed thet the `getint` function does something interesting to catch errors.

Rather than just saying

```
int getint(const string& prompt) {
  string ageword = getword(prompt);
  return stoi(ageword);
}
```

(which crashes the program if `ageword` does not contain an integer)

it does this:

```
int getint(const string& prompt) {
  while (true) {
    string ageword = getword(prompt);
    try {
      return stoi(ageword);
    } catch (invalid_argument& e) {
      cout << "Error: invalid input\n";
    }
  }
}
```

Catching errors using exceptions

- Exceptions can be used to catch unexpected events, like entering a non-number for age.

Dealing with errors

You may have noticed thet the `getint` function does something interesting to catch errors.

Rather than just saying

```
int getint(const string& prompt) {
  string ageword = getword(prompt);
  return stoi(ageword);
}
```

(which crashes the program if `ageword` does not contain an integer)

it does this:

```
int getint(const string& prompt) {
  while (true) {
    string ageword = getword(prompt);
    try {
      return stoi(ageword);
    } catch (invalid_argument& e) {
      cout << "Error: invalid input\n";
    }
  }
}
```

Catching errors using exceptions

- Exceptions can be used to catch unexpected events, like entering a non-number for age.
- This goes via the `try/catch` construct.

Dealing with errors

You may have noticed thet the `getint` function does something interesting to catch errors.

Rather than just saying

```
int getint(const string& prompt) {
  string ageword = getword(prompt);
  return stoi(ageword);
}
```

(which crashes the program if `ageword` does not contain an integer)

it does this:

```
int getint(const string& prompt) {
  while (true) {
    string ageword = getword(prompt);
    try {
      return stoi(ageword);
    } catch (invalid_argument& e) {
      cout << "Error: invalid input\n";
    }
  }
}
```

Catching errors using exceptions

- 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".

Dealing with errors

You may have noticed thet the `getint` function does something interesting to catch errors.

Rather than just saying

```
int getint(const string& prompt) {
  string ageword = getword(prompt);
  return stoi(ageword);
}
```

(which crashes the program if `ageword` does not contain an integer)

it does this:

```
int getint(const string& prompt) {
  while (true) {
    string ageword = getword(prompt);
    try {
      return stoi(ageword);
    } catch (invalid_argument& e) {
      cout << "Error: invalid input\n";
    }
  }
}
```

Catching errors using exceptions

- 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).

C++ Details

C++ Details: 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 i.e., truth value: `bool`

- array, pointer, class, structure, . . .

C++ Details: 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 i.e., truth value: `bool`

- array, pointer, class, structure, . . .

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;
```

C++ Details: 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 i.e., truth value: `bool`

- array, pointer, class, structure, ...

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;
```

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

C++ Details: 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 i.e., truth value: `bool`

- array, pointer, class, structure, ...

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;
```

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

const

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

C++ Details: Functions

Function = a piece of code that can be reused.

C++ Details: 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 specfic type

These three properties are called the function's signature.

C++ Details: 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 specfic 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;

C++ Details: 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 specfic 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.

C++ Details: 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 specfic 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++ function example

```cpp
// funcexample.cpp

// external function declarations:
#include <iostream>
#include <cmath>

// function declaration:
double geometric_mean(double a, double b);

// main function to call when program starts:
int main()
{
    double x = 16.3;
    double y = 102.4;
    std::cout << geometric_mean(x,y) << "\n";
}

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

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

$ module load gcc/12

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

$ ./funcexample
40.8549

$
```

C++ Details: Functions

- Function declaration (prototype/signature/interface)

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

`argument-spec` = comma separated list of variable definitions

C++ Details: 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 ;
}
```

C++ Details: 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`.

C++ Details: 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).

C++ Details: 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).
The function definition can double as the declaration if it preceeds all uses of it in the same source file.

C++ Details: 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).
The function definition can double as the declaration if it preceeds all uses of it in the same source file.

- Function call

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

`argument-list` = comma separated list of values

C++ Details: Scope

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

C++ Details: 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 code hits the closing curly brace (}) that correspond to the opening curly brace ({) that started the block. This is its local scope.

C++ Details: 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 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.

C++ Details: 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 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, variables from that block will not be known in the body of the function.

C++ Details: 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 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, variables 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.**

C++ Details: 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 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, variables 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.

C++ Details: Arguments by value or by reference

Passing function arguments by value

```
// passval.cpp
#include <iostream>
void inc(int i)
{
    i = i+1;
}

int main()
{
    int j = 10;
    inc(j);
    std::cout << j << "\n";
}
```

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

C++ Details: Arguments by value or by reference

Passing function arguments by value

```
// passval.cpp
#include <iostream>
void inc(int i)
{
    i = i+1;
}

int main()
{
    int j = 10;
    inc(j);
    std::cout << j << "\n";
}
```

```
$ g++ -std=c++17 -o passval passval.cpp
$ ./passval
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++ Details: Arguments by value or by reference

Passing function arguments by reference

```
// passref.cpp
#include <iostream>
void inc(int &i)
{
    i = i+1;
}

int main()
{
    int j = 10;
    inc(j);
    std::cout << j << "\n";
}
```

```
$ g++ -std=c++17 -o passref passref.cpp
$ ./passref
11
$
```

C++ Details: Arguments by value or by reference

Passing function arguments by reference

```
// passref.cpp
#include <iostream>
void inc(int &i)
{
    i = i+1;
}

int main()
{
    int j = 10;
    inc(j);
    std::cout << j << "\n";
}
```

```
$ g++ -std=c++17 -o passref passref.cpp
$ ./passref
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++ Details: 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

C++ Details: 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 are true (also: `a and b`)

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

C++ Details: 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 Substract 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`)

C++ Details: 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 Substract 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

C++ Details: What is 1/4?

```
1/4 = 0
```

C++ Details: What is 1/4?

`1/4 = 0`

Why?

- In 1/4 both operands, *i.e.*, 1 and 4, are integers.

C++ Details: 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.

C++ Details: 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.

C++ Details: 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.

C++ Details: 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.

The fix for `1/4 = 0`? Convert between types.

C++ Details: 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.

The fix for `1/4 = 0`? Convert between types.
In C/C++ this is called **type casting**.

Casting one numeric type into another

Use `static_cast<OTHERTYPE>(...)`

Casting one numeric type into another

Use static_cast<OTHERTYPE>(...)

Example:

```
// 1over4.cpp
#include <iostream>

int main()
{
    int   a = 1;
    int   b = 4;
    int   c = a/b;
    float d = static_cast<float>(a)
              / static_cast<float>(b);

    std::cout << c << " "
              << d << " "
              << static_cast<int>(d) << "\n";
}
```

Casting one numeric type into another

Use `static_cast<OTHERTYPE>(...)`

Example:

```
// 1over4.cpp
#include <iostream>

int main()
{
    int   a = 1;
    int   b = 4;
    int   c = a/b;
    float d = static_cast<float>(a)
              / static_cast<float>(b);

    std::cout << c << " "
              << d << " "
              << static_cast<int>(d) << "\n";
}
```

```
$ g++ -std=c++17 1over4.cpp -o 1over4

$ ./1over4
0 0.25 0
```

Casting one numeric type into another

Use `static_cast<OTHERTYPE>(...)`

Example:

```
// 1over4.cpp
#include <iostream>

int main()
{
    int   a = 1;
    int   b = 4;
    int   c = a/b;
    float d = static_cast<float>(a)
            / static_cast<float>(b);

    std::cout << c << " "
              << d << " "
              << static_cast<int>(d) << "\n";
}
```

```
$ g++ -std=c++17 1over4.cpp -o 1over4

$ ./1over4
0 0.25 0
```

Note: the older C++-style casting, `float(a)`, `int(d)`, etc. *still works, but less precise.*

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.

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. E.g in

```
#include <iostream>
double unchanged(int i) {
   return i;
}
int main() {
   std::cout << unchanged(2.3) << "\n";
}
```

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. E.g in

```
#include <iostream>
double unchanged(int i) {
   return i;
}
int main() {
   std::cout << unchanged(2.3) << "\n";
}
```

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

C++ Details: 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

C++ Details: 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, etc. of a module in a namespace:

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

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

- Effectively prefixes anything defines in ... with `modname::`

C++ Details: Namespaces

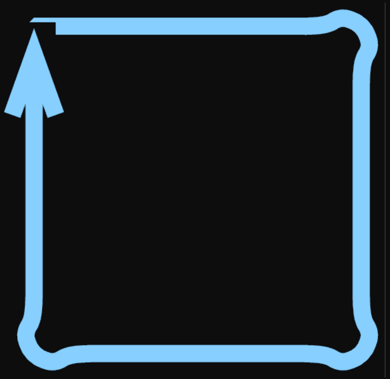
- Variables and function, as well as variable types, have names.
- In larger projects, you could have variable types of the same name.
- To avoid such name clashes, one can use namespaces
- One usually puts all functions, types, etc. of a module in a namespace:

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

  (namespace is the keyword, modname is an identifier of your choosing)
- Effectively prefixes anything defines in ... with `modname::`
- Many standard functions/types are in namespace std.
- You can make all things in a namespace available without the prefix with "`using namespace modname`". You can also make just one thing available, e.g.

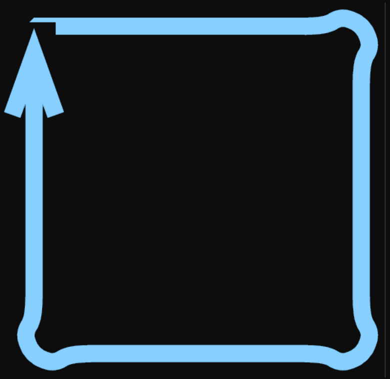
```
using std::cout;
cout << "Hello, world" << "\n";
```

C++ Details: Loops



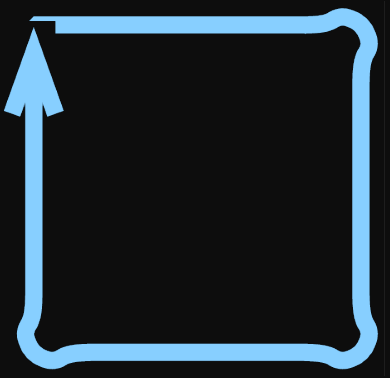
- In scientific computing, we often want to do the same thing for all points on a grid, or for every piece of experimental data, etc.

C++ Details: 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.

C++ Details: 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++ Details: Loops

Three forms:

- traditional for loop

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

- range-based for loop

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

- while loop

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

You can use the break statement to exit the loop.

Example

```
#include <iostream>
int main() {
    for (int i=1; i<=10; i++) {
        std::cout << i << " ";
    }
    std::cout << "\n";
}
```

```
#include <iostream>
int main() {
    for (int i: {1,2,3,4,5,6,7,8,9,10}) {
        std::cout << i << " ";
    }
    std::cout << "\n";
}
```

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

C++ Details: 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.
- The null pointer, denoted by `nullptr`, points to nowhere.

C++ Details: 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.
- The null pointer, denoted by `nullptr`, points to nowhere.

Definition:

```
type* name ;
```

Assignment ("take-address-of"):

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

Deferencing ("get-content-at-address"):

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

C++ Details: 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.
- The null pointer, denoted by `nullptr`, points to nowhere.

Definition:

```
type* name ;
```

Assignment ("take-address-of"):

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

Deferencing ("get-content-at-address"):

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

- Pointers are used for:
  - Arrays
  - Dynamic memory allocation
  - Linked lists, binary trees, ...
  - Calling functions written in C or Fortran

C++ Details: 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.
- The null pointer, denoted by `nullptr`, points to nowhere.

- Pointers are used for:
  - Arrays
  - Dynamic memory allocation
  - Linked lists, binary trees, . . .
  - Calling functions written in C or Fortran

Definition:

```
type* name ;
```

Assignment ("take-address-of"):

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

Deferencing ("get-content-at-address"):

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

Example:

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

Danger, Will Robinson!

These are so-called **raw pointers**, in contrast with **smart pointers** that are in the C++ standard library.
Raw pointers are considered dangerous. What could go wrong?

Danger, Will Robinson!

These are so-called **raw pointers**, in contrast with **smart pointers** that are in the C++ standard library. Raw pointers are considered dangerous. What could go wrong?

- Raw pointers support arithmetic, which can cause pointers to point to invalid or undefined memory.

```
int* addr;
int a = 1;
addr = &a;
*addr = 2; // fine, sets a to 2
addr += 1; // allowed, but undefined what addr points at
*addr = 3; // mayhem!
```

Danger, Will Robinson!

These are so-called **raw pointers**, in contrast with **smart pointers** that are in the C++ standard library. Raw pointers are considered dangerous. What could go wrong?

- Raw pointers support arithmetic, which can cause pointers to point to invalid or undefined memory.

```
int* addr;
int a = 1;
addr = &a;
*addr = 2; // fine, sets a to 2
addr += 1; // allowed, but undefined what addr points at
*addr = 3; // mayhem!
```

- Uninitialized pointers can point anywhere: using those can cause undefined mayhem too.

```
int* addr;
*addr = 1 ; // syntactically allowed, but undefined where in memory '1' is written.
```

Danger, Will Robinson!

These are so-called **raw pointers**, in contrast with **smart pointers** that are in the C++ standard library. Raw pointers are considered dangerous. What could go wrong?

- Raw pointers support arithmetic, which can cause pointers to point to invalid or undefined memory.

```
int* addr;
int a = 1;
addr = &a;
*addr = 2; // fine, sets a to 2
addr += 1; // allowed, but undefined what addr points at
*addr = 3; // mayhem!
```

- Uninitialized pointers can point anywhere: using those can cause undefined mayhem too.

```
int* addr;
*addr = 1 ; // syntactically allowed, but undefined where in memory '1' is written.
```

- Pointers can be used for resource management, but are susceptible to resource leaks; we'll cover this, and its solution, later.

Danger, Will Robinson!

These are so-called **raw pointers**, in contrast with **smart pointers** that are in the C++ standard library. Raw pointers are considered dangerous. What could go wrong?

- Raw pointers support arithmetic, which can cause pointers to point to invalid or undefined memory.

```
int* addr;
int a = 1;
addr = &a;
*addr = 2; // fine, sets a to 2
addr += 1; // allowed, but undefined what addr points at
*addr = 3; // mayhem!
```

- Uninitialized pointers can point anywhere: using those can cause undefined mayhem too.

```
int* addr;
*addr = 1 ; // syntactically allowed, but undefined where in memory '1' is written.
```

- Pointers can be used for resource management, but are susceptible to resource leaks; we'll cover this, and its solution, later.

- Pointers are used for arrays (next topic), and can give access beyond the array's end.r

C++ Details: 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++ Details: 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 << "\n";
}
```

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

C++ Details: Automatic arrays, example

```cpp
// 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 << "\n";
}
```

```
$ 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++ Details: 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 << "\n";
}
```

```
$ 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++ Details: Dynamically allocated array (raw)

Dynamically allocated arrays are accessed using a pointer to memory:

```
type* name ;
```

They can be allocated using the keyword new :

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

(the square brackets are part of the syntax)

and deallocated with the `delete` statement:

```
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.
- `name` has no idea of its size.

Dynamic arrays - Improved version of the example

```cpp
// 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 << "\n";
    delete[] a;
}
```

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

Dynamic arrays - Improved version of the example

```
// 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 << "\n";
    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++.

Dynamic arrays - Improved version of the example

```
// 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 << "\n";
    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.

C++ Details: 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 << "\n";
    delete v;
}
```

Note the absence of [] in the delete statement.

You might use this in more dynamic data structures.

C++ Details: 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 << "\n";
    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 << "\n";
    // no delete necessary
}
```

C++ Details: 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 << "\n";
}
```

C++ Details: 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 << "\n";
}
```

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

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

C++ Details: 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 << "\n";
}
```

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.

C++ Details: 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 << "\n";
}
```

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 seconds argument, is converted to a pointer to the first element of the array.

C++ Details: 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 << "\n";
}
```

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 seconds argument, is converted to a pointer to the first element of the array.

From this point on, there is no other way to deduce how big the array was.

C++ Details: Command Line Arguments

Linux commands can be followed by arguments.

C++ Details: 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

C++ Details: 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

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`.

C++ Details: Command Line Arguments Example

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

C++ Details: Command Line Arguments Example

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

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

C++ Details: Exceptions

Syntax:

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

C++ Details: Exceptions

Syntax:

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

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

C++ Details: Exceptions

Syntax:

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

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

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

C++ Details: Exceptions

Syntax:

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

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

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

Object-oriented programming and templates

C++ Overview: Using classes and 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: Using classes and objects

Example of a member function/method

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

C++ Overview: Using 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;
```

C++ Overview: Using 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?

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.

Templates

Usage

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

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

Templates

Usage

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

```
tmplcls<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)
```

Scope revisited for objects

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

In addition, when going out of scope, a special member function of the called the destructor is called. This gives objects that dynamically allocate memory the opportunity to delete that memory.

This is how `std::unique_ptr` and `std::shared_ptr` work.

Dynamic allocation revisited using smart pointers

Dynamically allocated arrays can also be defined as a smart pointer to memory:

```
#include <memory>
std::shared_ptr<type[]> sarr ; // can be shared by copying
std::unique_ptr<type[]> uarr ; // cannot be shared
```

Dynamic allocation revisited using smart pointers

Dynamically allocated arrays can also be defined as a smart pointer to memory:

```
#include <memory>
std::shared_ptr<type[]> sarr ; // can be shared by copying
std::unique_ptr<type[]> uarr ; // cannot be shared
```

Allocated as follows:

```
uarr = std::unique_ptr<type[]>(new type[number]);
uarr = std::make_unique<type[]>(number);
sarr = std::shared_ptr<type[]>(new type[number]);
sarr = std::make_shared<type[]>(number);  // only in C++20
```

Dynamic allocation revisited using smart pointers

Dynamically allocated arrays can also be defined as a smart pointer to memory:

```
#include <memory>
std::shared_ptr<type[]> sarr ; // can be shared by copying
std::unique_ptr<type[]> uarr ; // cannot be shared
```

Allocated as follows:

```
uarr = std::unique_ptr<type[]>(new type[number]);
uarr = std::make_unique<type[]>(number);
sarr = std::shared_ptr<type[]>(new type[number]);
sarr = std::make_shared<type[]>(number);  // only in C++20
```

- Memory is automatically deallocated when pointer goes out of scope (and no copies are left)!
- No pointer arithmetic allowed!
- Usage of these arrays is the same as for automatic arrays.
- Can access all available memory.
- But these smart arrays still have no idea of their size.
- So can still access beyond end of array with `sarr[i]`, `uarr[i]` if `i>=number`.

Array allocation - Smart version

```cpp
// smartarr.cpp

#include <memory>
#include <iostream>

int main()
{
  std::unique_ptr<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 << "\n";
}
```

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

Variable definitions revisited: auto

Every variable must be defined and in that definition, has to be declared as a specific type.
But that sometimes means you have to mention the type several times, e.g.

```
std::unique_ptr<int[]> a;
a = std::unique_ptr<int[]>(new int[6]);
```

The type **int[]** is specified 3 times, and has to be the same in all three spots.

Variable definitions revisited: auto

Every variable must be defined and in that definition, has to be declared as a specific type.
But that sometimes means you have to mention the type several times, e.g.

```
std::unique_ptr<int[]> a;
a = std::unique_ptr<int[]>(new int[6]);
```

The type **int[]** is specified 3 times, and has to be the same in all three spots.

Combine declaration and initialization

You should combine declaration and initialization whenever possible, so the above can become:

```
std::unique_ptr<int[]> a(new int[6]);
```

Variable definitions revisited: auto

Every variable must be defined and in that definition, has to be declared as a specific type.
But that sometimes means you have to mention the type several times, e.g.

```
std::unique_ptr<int[]> a;
a = std::unique_ptr<int[]>(new int[6]);
```

The type **int[]** is specified 3 times, and has to be the same in all three spots.

Combine declaration and initialization

You should combine declaration and initialization whenever possible, so the above can become:

```
std::unique_ptr<int[]> a(new int[6]);
```

When initialization value determines type, use auto

When you combine variable declaration with initialization, if the C++ compiler can deduce the variable type from the initialization value, you may replace the type specification with the `auto` keyword.

```
auto a = std::make_unique_ptr<int[]>(6);
```

Auto caution

While it is tempting to always use `auto`, for **numerical types**, declare the variable types **explicitly**.

Auto caution

While it is tempting to always use **`auto`**, for **numerical types**, declare the variable types **explicitly**.

E.g., **do not** replace code like this:

```
double x = 1;
double y = 0.5;
x += y;
```

with

```
auto x = 1;
auto y = 0.5;
x += y;
```

In this case, `x` will have the wrong value (can you see why?)

Auto caution

While it is tempting to always use `auto`, for **numerical types**, declare the variable types **explicitly**.

E.g., **do not** replace code like this:

```
double x = 1;
double y = 0.5;
x += y;
```

with

```
auto x = 1;
auto y = 0.5;
x += y;
```

In this case, `x` will have the wrong value (can you see why?)

Tip: Be explicit about numerical and other basic types

Auto caution

While it is tempting to always use `auto`, for **numerical types**, declare the variable types **explicitly**.

E.g., **do not** replace code like this:

```
double x = 1;
double y = 0.5;
x += y;
```

with

```
auto x = 1;
auto y = 0.5;
x += y;
```

In this case, `x` will have the wrong value (can you see why?)

Tip: Be explicit about numerical and other basic types

Furthermore, if the initializing expression does not have a type that is obvious to the programmer, don't use `auto`.

Auto caution

While it is tempting to always use `auto`, for **numerical types**, declare the variable types **explicitly**.

E.g., **do not** replace code like this:

```
double x = 1;
double y = 0.5;
x += y;
```

with

```
auto x = 1;
auto y = 0.5;
x += y;
```

In this case, `x` will have the wrong value (can you see why?)

Tip: Be explicit about numerical and other basic types

Furthermore, if the initializing expression does not have a type that is obvious to the programmer, don't use `auto`. So never:

```
auto a = f();
```

Libraries in C++

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.

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

Standard Library Example: Sort an array

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

int main()
{
   std::unique_ptr<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] << "\n";
   }
}
```

- 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++ IO Standard Library

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

These are defined in the header file `iostream`.

C++ IO Standard Library Example

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

C++ File IO Standard Library

- 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++ File IO Standard Library Examples

Writing to a file

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

C++ File IO Standard Library Examples

Writing to a file

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

Reading from a file

```
#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 <<"\n";
}
```

The only way to learn a language is to use it.

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