

COMP110: Principles of Computing

11: Further C++

DEADLINES

Representing numbers



Powers of 10

Powers of 10

$$10^6 = 1 \underbrace{000000}_{6 \text{ zeroes}}$$

Powers of 10

$$10^6 = 1 \underbrace{000000}_{6 \text{ zeroes}}$$

$$10^1 = 10$$

Powers of 10

$$10^6 = 1 \underbrace{000000}_{6 \text{ zeroes}}$$

$$10^1 = 10$$

$$10^0 = 1$$

Powers of 10

$$10^6 = 1 \underbrace{000000}_{6 \text{ zeroes}}$$

$$10^1 = 10$$

$$10^0 = 1$$

$$10^{-1} = 0.1$$

Powers of 10

$$10^6 = 1 \underbrace{000000}_{6 \text{ zeroes}}$$

$$10^1 = 10$$

$$10^0 = 1$$

$$10^{-1} = 0.1$$

$$10^{-6} = 0.\underbrace{000000}_{5 \text{ zeroes}}1$$

Scientific notation

Scientific notation

- ▶ A way of writing **very large** and **very small** numbers

Scientific notation

- ▶ A way of writing **very large** and **very small** numbers
- ▶ $a \times 10^b$, where

Scientific notation

- ▶ A way of writing **very large** and **very small** numbers
- ▶ $a \times 10^b$, where
 - ▶ a ($1 \leq |a| < 10$) is the **mantissa**

Scientific notation

- ▶ A way of writing **very large** and **very small** numbers
- ▶ $a \times 10^b$, where
 - ▶ a ($1 \leq |a| < 10$) is the **mantissa**
 - ▶ (a is a positive or negative number with a single non-zero digit before the decimal point)

Scientific notation

- ▶ A way of writing **very large** and **very small** numbers
- ▶ $a \times 10^b$, where
 - ▶ a ($1 \leq |a| < 10$) is the **mantissa**
 - ▶ (a is a positive or negative number with a single non-zero digit before the decimal point)
 - ▶ b (an integer) is the **exponent**

Scientific notation

- ▶ A way of writing **very large** and **very small** numbers
- ▶ $a \times 10^b$, where
 - ▶ a ($1 \leq |a| < 10$) is the **mantissa**
 - ▶ (a is a positive or negative number with a single non-zero digit before the decimal point)
 - ▶ b (an integer) is the **exponent**
- ▶ E.g. 1 light year = 9.461×10^{15} metres

Scientific notation

- ▶ A way of writing **very large** and **very small** numbers
- ▶ $a \times 10^b$, where
 - ▶ a ($1 \leq |a| < 10$) is the **mantissa**
 - ▶ (a is a positive or negative number with a single non-zero digit before the decimal point)
 - ▶ b (an integer) is the **exponent**
- ▶ E.g. 1 light year = 9.461×10^{15} metres
- ▶ E.g. Planck's constant = 6.626×10^{-34} joules

Scientific notation

- ▶ A way of writing **very large** and **very small** numbers
- ▶ $a \times 10^b$, where
 - ▶ a ($1 \leq |a| < 10$) is the **mantissa**
 - ▶ (a is a positive or negative number with a single non-zero digit before the decimal point)
 - ▶ b (an integer) is the **exponent**
- ▶ E.g. 1 light year = 9.461×10^{15} metres
- ▶ E.g. Planck's constant = 6.626×10^{-34} joules
- ▶ Socrative FALCOMPED

Scientific notation in C++

Scientific notation in C++

Instead of writing $\times 10$, write e

Scientific notation in C++

Instead of writing $\times 10$, write e

```
double lightYear = 9.461e15;  
double plancksConstant = 6.626e-34;
```

Scientific notation in C++

Instead of writing $\times 10$, write e

```
double lightYear = 9.461e15;  
double plancksConstant = 6.626e-34;
```

This also works in Python and many other programming languages

Floating point numbers

Floating point numbers

- ▶ Similar to scientific notation, but **base 2** (binary)

Floating point numbers

- ▶ Similar to scientific notation, but **base 2** (binary)
- ▶ $\pm \text{mantissa} \times 2^{\text{exponent}}$

Floating point numbers

- ▶ Similar to scientific notation, but **base 2** (binary)
- ▶ $\pm \text{mantissa} \times 2^{\text{exponent}}$
- ▶ Sign is stored as a single bit: 0 = +, 1 = -

Floating point numbers

- ▶ Similar to scientific notation, but **base 2** (binary)
- ▶ $\pm \text{mantissa} \times 2^{\text{exponent}}$
- ▶ Sign is stored as a single bit: 0 = +, 1 = -
- ▶ Mantissa is a binary number with a 1 before the point; only the digits after the point are stored

Floating point numbers

- ▶ Similar to scientific notation, but **base 2** (binary)
- ▶ $\pm \text{mantissa} \times 2^{\text{exponent}}$
- ▶ Sign is stored as a single bit: 0 = +, 1 = -
- ▶ Mantissa is a binary number with a 1 before the point; only the digits after the point are stored
- ▶ Exponent is a signed integer, stored with a **bias**

IEEE 754 floating point formats

Type	Sign	Exponent	Mantissa	Total
<code>float</code>	1 bit	8 bits	23 bits	32 bits
<code>double</code>	1 bit	11 bits	52 bits	64 bits

IEEE 754 floating point formats

Type	Sign	Exponent	Mantissa	Total
<code>float</code>	1 bit	8 bits	23 bits	32 bits
<code>double</code>	1 bit	11 bits	52 bits	64 bits

Exponent is stored with a **bias**:

- ▶ Single precision: store exponent + 127
- ▶ Double precision: store exponent + 1023

Example

Example

0 10000001 101000000000000000000000

Example

0 10000001 101000000000000000000000

► Exponent: $129 - 127 = 2$

Example

0 10000001 101000000000000000000000

- ▶ Exponent: $129 - 127 = 2$
- ▶ Mantissa: binary 1.101

Example

0 10000001 101000000000000000000000

- ▶ Exponent: $129 - 127 = 2$
- ▶ Mantissa: binary 1.101
- ▶ $1 + \frac{1}{2} + \frac{1}{8} = 1.625$

Example

0 10000001 101000000000000000000000

- ▶ Exponent: $129 - 127 = 2$
- ▶ Mantissa: binary 1.101
- ▶ $1 + \frac{1}{2} + \frac{1}{8} = 1.625$
- ▶ $1.625 \times 2^2 = 6.5$

Example

0 10000001 101000000000000000000000

- ▶ Exponent: $129 - 127 = 2$
- ▶ Mantissa: binary 1.101
- ▶ $1 + \frac{1}{2} + \frac{1}{8} = 1.625$
- ▶ $1.625 \times 2^2 = 6.5$
- ▶ Alternatively: $1.101 \times 2^2 = 110.1$

Example

0 10000001 101000000000000000000000

- ▶ Exponent: $129 - 127 = 2$
- ▶ Mantissa: binary 1.101
- ▶ $1 + \frac{1}{2} + \frac{1}{8} = 1.625$
- ▶ $1.625 \times 2^2 = 6.5$
- ▶ Alternatively: $1.101 \times 2^2 = 110.1$
- ▶ $= 4 + 2 + \frac{1}{2} = 6.5$

Socratic FALCOMPED

Socratic FALCOMPED

What is the value of this number expressed in IEEE 754 single precision format?

0 01111100 100110000000000000000000

You have **5 minutes**, and you **may** use a calculator!

Floating point numbers

Floating point numbers

- ▶ Similar to scientific notation, but **base 2** (binary)

Floating point numbers

- ▶ Similar to scientific notation, but **base 2** (binary)
- ▶ $\pm \text{mantissa} \times 2^{\text{exponent}}$

Floating point numbers

- ▶ Similar to scientific notation, but **base 2** (binary)
- ▶ $\pm \text{mantissa} \times 2^{\text{exponent}}$
- ▶ Sign is stored as a single bit: 0 = +, 1 = -

Floating point numbers

- ▶ Similar to scientific notation, but **base 2** (binary)
- ▶ $\pm \text{mantissa} \times 2^{\text{exponent}}$
- ▶ Sign is stored as a single bit: 0 = +, 1 = -
- ▶ Mantissa is a binary number with a 1 before the point; only the digits after the point are stored

Floating point numbers

- ▶ Similar to scientific notation, but **base 2** (binary)
- ▶ $\pm \text{mantissa} \times 2^{\text{exponent}}$
- ▶ Sign is stored as a single bit: 0 = +, 1 = -
- ▶ Mantissa is a binary number with a 1 before the point; only the digits after the point are stored
- ▶ Exponent is a signed integer, stored with a **bias**

Limitations of floating point numbers

Limitations of floating point numbers

- ▶ Precision **varies** by **magnitude**: numbers near 0 can be stored more accurately than numbers further from 0.

Limitations of floating point numbers

- ▶ Precision **varies** by **magnitude**: numbers near 0 can be stored more accurately than numbers further from 0.
 - ▶ Why? Socratic FALCOMPED

Limitations of floating point numbers

- ▶ Precision **varies** by **magnitude**: numbers near 0 can be stored more accurately than numbers further from 0.
 - ▶ Why? Socratic `FALCOMPED`
- ▶ Many numbers cannot be represented exactly, e.g. $\frac{1}{5}$

Limitations of floating point numbers

- ▶ Precision **varies** by **magnitude**: numbers near 0 can be stored more accurately than numbers further from 0.
 - ▶ Why? Socratic `FALCOMPED`
- ▶ Many numbers cannot be represented exactly, e.g. $\frac{1}{5}$
 - ▶ Similar to how decimal notation cannot exactly represent $\frac{1}{3} = 0.333333 \dots$

Limitations of floating point numbers

- ▶ Precision **varies** by **magnitude**: numbers near 0 can be stored more accurately than numbers further from 0.
 - ▶ Why? Socratic `FALCOMPED`
- ▶ Many numbers cannot be represented exactly, e.g. $\frac{1}{5}$
 - ▶ Similar to how decimal notation cannot exactly represent $\frac{1}{3} = 0.333333 \dots$
- ▶ This can lead to **rounding errors** with some calculations

Limitations of floating point numbers

- ▶ Precision **varies** by **magnitude**: numbers near 0 can be stored more accurately than numbers further from 0.
 - ▶ Why? Socratic `FALCOMPED`
- ▶ Many numbers cannot be represented exactly, e.g. $\frac{1}{5}$
 - ▶ Similar to how decimal notation cannot exactly represent $\frac{1}{3} = 0.333333 \dots$
- ▶ This can lead to **rounding errors** with some calculations
 - ▶ E.g. according to Python,
`0.1 + 0.2 == 0.30000000000000004`

Testing for equality

Testing for equality

- ▶ Due to rounding errors, using `==` or `!=` with floating point numbers is almost always a bad idea

Testing for equality

- ▶ Due to rounding errors, using `==` or `!=` with floating point numbers is almost always a bad idea
- ▶ E.g. in Python, `0.1 + 0.2 == 0.3` evaluates to `False`

Testing for equality

- ▶ Due to rounding errors, using `==` or `!=` with floating point numbers is almost always a bad idea
- ▶ E.g. in Python, `0.1 + 0.2 == 0.3` evaluates to `False`
- ▶ Better to check for **approximate equality**: calculate the difference between the numbers, and check that it's smaller than some threshold

Modular program design



Modular program design

- ▶ We saw previously that **splitting your code into several files** is generally a good idea

Modular program design

- ▶ We saw previously that **splitting your code into several files** is generally a good idea
- ▶ Python makes it easy: any .py file can be **imported** on demand

Modular program design

- ▶ We saw previously that **splitting your code into several files** is generally a good idea
- ▶ Python makes it easy: any .py file can be **imported** on demand
- ▶ C++ is a little trickier...

Definitions and declarations

A function **definition** specifies its name, return type, parameters, and the code it contains:

```
double average(double n1, double n2)
{
    return (n1 + n2) / 2.0;
}
```

Definitions and declarations

A function **definition** specifies its name, return type, parameters, and the code it contains:

```
double average(double n1, double n2)
{
    return (n1 + n2) / 2.0;
}
```

A function **declaration** specifies everything **except** the code:

```
double average(double n1, double n2);
```

Definitions and declarations

A function **definition** specifies its name, return type, parameters, and the code it contains:

```
double average(double n1, double n2)
{
    return (n1 + n2) / 2.0;
}
```

A function **declaration** specifies everything **except** the code:

```
double average(double n1, double n2);
```

A declaration tells the compiler that this function exists, but is defined **elsewhere**

Sources and headers

- ▶ A C++ project contains two main types of file

Sources and headers

- ▶ A C++ project contains two main types of file
- ▶ **Source files** (.cpp) usually contain **definitions**

Sources and headers

- ▶ A C++ project contains two main types of file
- ▶ **Source files** (.cpp) usually contain **definitions**
- ▶ **Header files** (.h) usually contain **declarations**

Sources and headers

- ▶ A C++ project contains two main types of file
- ▶ **Source files** (.cpp) usually contain **definitions**
- ▶ **Header files** (.h) usually contain **declarations**
- ▶ For example, `myfile.cpp` may contain some function definitions, and `myfile.h` may contain the declarations for those functions

Sources and headers

- ▶ A C++ project contains two main types of file
- ▶ **Source files** (.cpp) usually contain **definitions**
- ▶ **Header files** (.h) usually contain **declarations**
- ▶ For example, `myfile.cpp` may contain some function definitions, and `myfile.h` may contain the declarations for those functions
- ▶ (Yep, that means you have to type the same thing twice in two different files...)

Example

words.cpp

```
void readWords()  
{  
    // code omitted  
}  
  
std::string chooseRandomWord()  
{  
    // code omitted  
}
```

words.h

```
#pragma once  
  
void readWords();  
std::string chooseRandomWord();
```

Example from last week

- ▶ `readWords()` and `chooseRandomWord()` are **defined** in `words.cpp`

Example from last week

- ▶ `readWords()` and `chooseRandomWord()` are **defined** in `words.cpp`
- ▶ `readWords()` and `chooseRandomWord()` are **declared** in `words.h`

Example from last week

- ▶ `readWords()` and `chooseRandomWord()` are **defined** in `words.cpp`
- ▶ `readWords()` and `chooseRandomWord()` are **declared** in `words.h`
- ▶ Any file which does `#include "words.h"` can call these functions as if they were declared in that file

How #include works

- ▶ `#include` works **exactly** as if the `#included` file were copied and pasted at the point where the `#include` directive appears

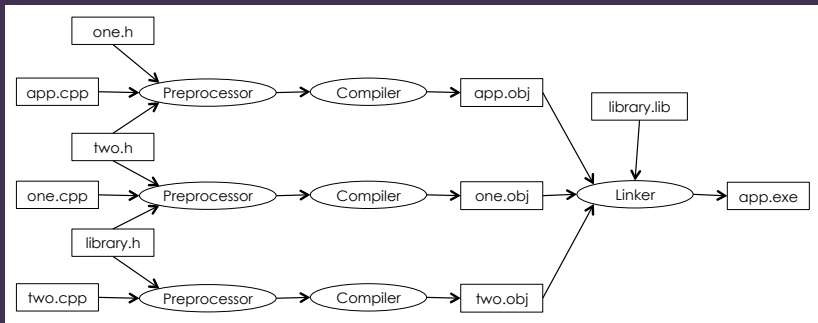
How `#include` works

- ▶ `#include` works **exactly** as if the `#included` file were copied and pasted at the point where the `#include` directive appears
- ▶ All header files should start with `#pragma once` — otherwise, `#include`ing the same file more than once will result in duplicate declaration errors

How `#include` works

- ▶ `#include` works **exactly** as if the `#included` file were copied and pasted at the point where the `#include` directive appears
- ▶ All header files should start with `#pragma once` — otherwise, `#include`ing the same file more than once will result in duplicate declaration errors
- ▶ Putting an `#include` directive in the wrong place (e.g. inside a function) will result in weird compile errors

The C++ build process



Pointers



Pointers

- ▶ A **pointer** is the address of a memory location

Pointers

- ▶ A **pointer** is the address of a memory location
- ▶ If T is a type, T^* is the type “pointer to T ”

Pointers

- ▶ A **pointer** is the address of a memory location
- ▶ If T is a type, T^* is the type “pointer to T ”
- ▶ $\&$ is the **address-of** operator: gets a pointer to something

Pointers

- ▶ A **pointer** is the address of a memory location
- ▶ If T is a type, T^* is the type “pointer to T ”
- ▶ $\&$ is the **address-of** operator: gets a pointer to something
- ▶ $*$ is the **dereference** operator: gets the thing the pointer points to

Classes in C++

```
class MyClass
{
public:
    MyClass() { /* constructor */ }
    ~MyClass() { /* destructor */ }

    void myMethod();
    int  anotherMethod(float foo);

    int  myField;
    bool anotherField
};
```

Allocating objects on the stack

Allocating objects on the stack

```
// Calls a parameterless constructor  
MyClass instance;  
  
// Calls a constructor with parameters  
MyClass otherInstance(1, 2, 3);
```

Allocating objects on the stack

```
// Calls a parameterless constructor  
MyClass instance;  
  
// Calls a constructor with parameters  
MyClass otherInstance(1, 2, 3);
```

Beware though — these instances are **destroyed** when the variable goes out of scope!

Allocating objects on the heap

- Objects can be allocated on the heap using the `new` keyword

Allocating objects on the heap

- ▶ Objects can be allocated on the heap using the `new` keyword
- ▶ `new` gives a pointer to the new instance

Allocating objects on the heap

- ▶ Objects can be allocated on the heap using the **new** keyword
- ▶ **new** gives a pointer to the new instance

```
// To use a parameterless constructor
```

```
MyClass* myInstance = new MyClass;
```

```
// To use a constructor with parameters
```

```
MyClass* myOtherInstance = new MyClass(1, 2, 3);
```

Deleting objects from the heap

- ▶ Objects instantiated with `new` must be deleted using `delete`

Deleting objects from the heap

- ▶ Objects instantiated with `new` must be deleted using `delete`

```
delete myInstance;
```

Deleting objects from the heap

- ▶ Objects instantiated with `new` must be deleted using `delete`

```
delete myInstance;
```

- ▶ Forgetting to do this is a **memory leak**

Deleting objects from the heap

- ▶ Objects instantiated with `new` must be deleted using `delete`

```
delete myInstance;
```

- ▶ Forgetting to do this is a **memory leak**
- ▶ Deleting something **twice** is bad

Deleting objects from the heap

- ▶ Objects instantiated with `new` must be deleted using `delete`

```
delete myInstance;
```

- ▶ Forgetting to do this is a **memory leak**
- ▶ Deleting something **twice** is bad
- ▶ Trying to **dereference a deleted pointer** is bad

Deleting objects from the heap

- ▶ Objects instantiated with `new` must be deleted using `delete`

```
delete myInstance;
```

- ▶ Forgetting to do this is a **memory leak**
- ▶ Deleting something **twice** is bad
- ▶ Trying to **dereference a deleted pointer** is bad
- ▶ Key concept is **ownership**: you're responsible for deleting it **if and only if** you own it

Addressing and dereferencing

```
int a = 7;  
  
// Address-of operator  
int* b = &a;  
  
// Dereferencing  
int c = *b;
```


Addressing and dereferencing

```
int a = 7;  
  
// Address-of operator  
int* b = &a;  
  
// Dereferencing  
int c = *b;
```

- ▶ `&` gets the **address** of a variable, i.e. a pointer to it
- ▶ `*` **dereferences** the pointer, i.e. looks up the thing it points to

Socratic FALCOMPED

```
int a = 7;  
int* b = &a;  
int c = *b;
```

Suppose that the variables are assigned to the following memory addresses:

Variable	a	b	c
Address	1000	1004	1008

Socratic FALCOMPED

```
int a = 7;  
int* b = &a;  
int c = *b;
```

Suppose that the variables are assigned to the following memory addresses:

Variable	a	b	c
Address	1000	1004	1008

1. What is the value of a ?

Socratic FALCOMPED

```
int a = 7;  
int* b = &a;  
int c = *b;
```

Suppose that the variables are assigned to the following memory addresses:

Variable	a	b	c
Address	1000	1004	1008

1. What is the value of a ?
2. What is the value of b ?

Socratic FALCOMPED

```
int a = 7;  
int* b = &a;  
int c = *b;
```

Suppose that the variables are assigned to the following memory addresses:

Variable	a	b	c
Address	1000	1004	1008

1. What is the value of a ?
2. What is the value of b ?
3. What is the value of c ?

Null pointer

- Pointers can have a special value `nullptr`

Null pointer

- ▶ Pointers can have a special value `nullptr`
- ▶ This signifies the pointer doesn't point to anything

Null pointer

- ▶ Pointers can have a special value `nullptr`
- ▶ This signifies the pointer doesn't point to anything

```
MyClass* notAnInstance = nullptr;
```


Null pointer

- ▶ Pointers can have a special value `nullptr`
- ▶ This signifies the pointer doesn't point to anything

```
MyClass* notAnInstance = nullptr;
```

- ▶ Similar to `None` in Python

Null pointer

- ▶ Pointers can have a special value `nullptr`
- ▶ This signifies the pointer doesn't point to anything

```
MyClass* notAnInstance = nullptr;
```

- ▶ Similar to `None` in Python
- ▶ You may also see `NULL` used instead of `nullptr` — the meaning is the same