COMP 3522

Object Oriented Programming in C++

Week 1

Agenda

1. Introduction
2. Toolchain
3. C++ program structure
4. Fundamental types

COMP 3522

INTRODUCTION

Me

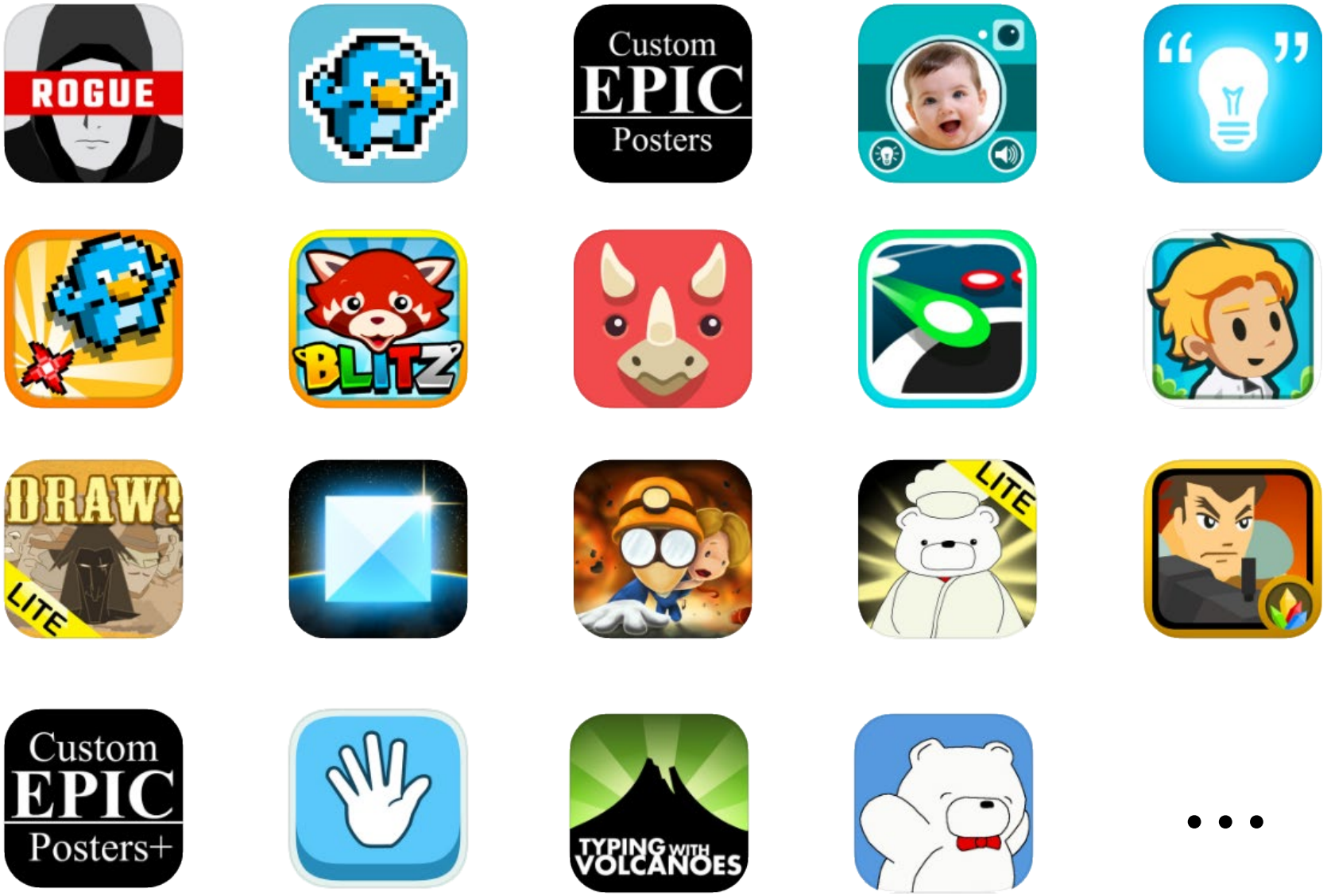
- Jeffrey Yim (call me **Jeff**)
- Email:
  - **jyim3@bcit.ca**
  - **Subject line [COMP3522]**
- Office Hours
  - By appointment on Discord
  - Mondays: 10:30am-2:20pm

Me

- Education:
  - Started at BCIT! w/Pascal
  - Queen's University
  - Bachelor's/Master's Computer Science
- Interests:
  - Game development (COMP 7051)
  - Unity
  - Tesla
  - Investing, Urban kiz, DJing
- Favorite language
  - C#, Python is growing on me

Industry games

iOS games


ROGUE
Custom
EPIC
Posters
BLITZ
DRAW!
LITE
LITE
Custom
EPIC
Posters+
TYPING WITH
VOLCANOES

...

COMP 3522

Three pillars of this course:

1. Learn how to **code in C++**:
    - Inheritance
    - Polymorphism
    - Exceptions
    - the Standard Template Library.

2. Learn how to **write good, clean, reusable code** using the S.O.L.I.D principles

3. Learn how to **write Design Patterns** – Behavioral, Creational, Structural patterns

Learning Outcomes

- Design, implement, debug and test intermediate to complex **object-oriented programs** in a modern high-level object-oriented programming language.
- Demonstrate familiarity with advanced **design idioms** including Dependency-inversion Principle, Liskov Substitution Principle, Open/Closed Principle, Interface-segregation Principle, Law of Demeter and Principle of Least Knowledge in a modern high-level object-oriented programming language.
- Use profiling tools to monitor the memory and CPU usage of programs.
- Use **multiple inheritance** to model complex abstractions.
- Use **exception handling** to catch errors and properly release resources.
- Demonstrate an understanding of advanced **design patterns** including Abstract Factory, Builder, Lazy Initialization, Bridge, Decorator, Facade, Proxy, Chain of Responsibility, Iterator, Mediator, State, and Strategy.
- Represent analysis and design models using sequence, collaboration and **class diagrams**.
- Develop systems using advanced architectures that **reduce system coupling** including dependency injection, asynchronous message passing, and observers.

Evaluation

- Your grading scheme:
  - Labs 10%
  - Quizzes (during labs) 10%
  - Assignments 20%
  - Midterm 30%
  - Final Exam 30%
- **You must pass the average of final+midterm to pass the course**
- A passing grade is 50.0%

Schedule

- **Lectures**
  - **Thursday 2:30pm-5:20pm**

Pre-recorded lectures are for review. Not necessary to watch them

- Each topic split into its own video

Schedule

- **Labs**
- Try your best to be punctual
- Time to work on lab and ask me questions
- Attendance is taken during labs
  - I need to report unapproved absences
  - Unapproved **absence of 10% or more** of the labs may result in **failure or forced withdrawal** from this course
- Due end of day Friday 11:59pm

Schedule

- **Labs & assignments**
- Labs to be completed individually, assignments can be completed in pairs
- No late submissions
- Submit code to github classroom

- Michael (Mike on discord) is our marker this term
- Direct specific mark questions to him

Schedule

- **Midterm & Final exam**
- In-person exam
- Best way to practice is complete labs, assignments, and exercises in lectures
- More information on topics and format later

Resources: Texts

1. The C++ Programming Language 4th Edition
2. The C++ Primer Plus 6th Edition

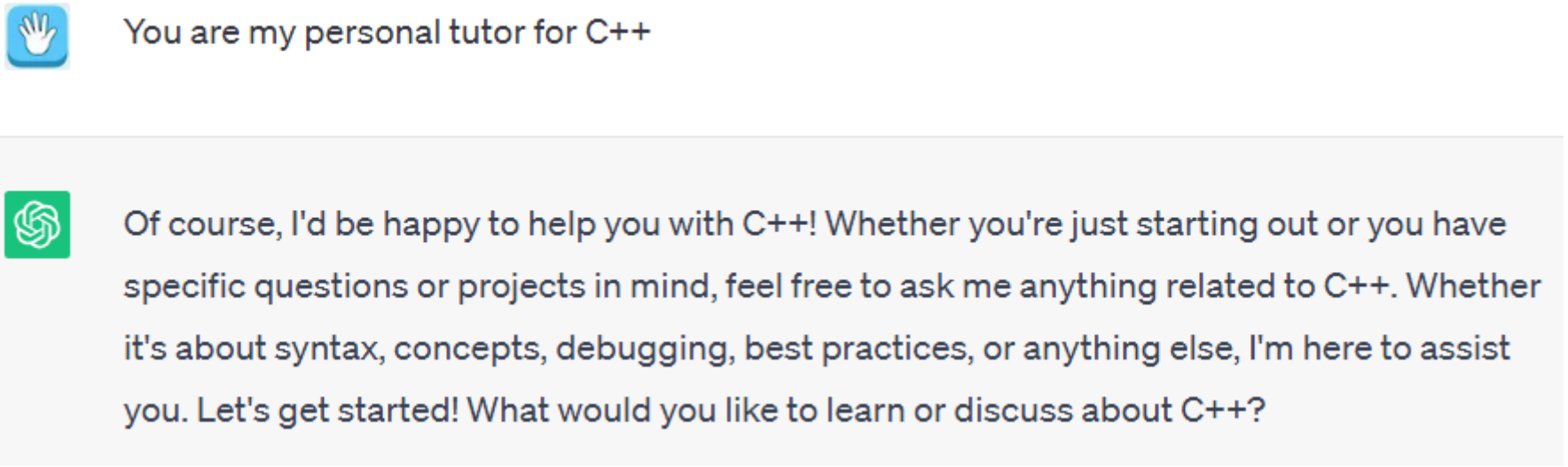
These are optional (but great to leaf through)

Resources: Online

1. http://en.cppreference.com
2. http://www.cplusplus.com/
3. https://stackoverflow.com/questions/388242/the-definitive-c-book-guide-and-list
4. https://isocpp.org/

5. https://www.tutorialspoint.com/cplusplus/
6. https://www.geeksforgeeks.org/c-plus-plus/

Resources: ChatGPT / Gen AI



- Use Gen AI as your personal tutor to explain concepts and show sample code. *But don't always trust it*
- Do not ask it to complete lab or assignment work for you
- Do not submit Gen AI code as your own

Lecture Breakdown

- Explaining topic and theory
- Examine code
- Q&A discussions
- 10 minute break every 50 minutes

- Optional review videos: each topic is a separate video

The Learning Hub

- used for grades and posting lecture slides/labs/assignments

Github classroom

- used for submitting labs/assignments

Academic Conduct

- Come to class
  - Topic introductions, announcements and reminders of due dates are given at the beginning of lecture
- Quizzes during beginning of labs (starting Week 2)
  - On learning hub
  - Multiple choice, true/false, matching, etc
- **Put phone on silent and put it away**
- Be respectful and kind

Collaboration and Plagiarism

- You are encouraged to collaborate by:
  - Completing optional lecture exercises
  - Helping each other understand material and assignments
  - Discussing requirements and approaches to problems
- What's not allowed:
  - Exchanging or sharing code snippets/solutions
  - Submitting someone else's work as your own
- Academic Integrity policy
  www.bcit.ca/files/pdf/policies/5104.pdf

TOOLCHAIN

Toolchain

1. **Communicate with Discord**
   - Best place to ask for help
   - Can have private conversations with me or with each other
   - It is where I will have office hours, share news, info about the course
   - **Discord link on the learning hub**
2. We will use **Github classrom** this term for submitting assignments and labs
3. **Backup**: Submit labs/assignments with **Student Developer Pack from Github**
   - Unlimited private repositories
   - **https://education.github.com/pack**

Toolchain

3. **IDE (integrated development environment)**
   - **CLion** (free for students from JetBrains)
     - FREE
     - Supports C++20 and unit testing
     - We are all using **C++20** this term
   - Backup - Visual Studio 2017 Community or Enterprise
     - FREE for BCIT students at **https://www.bcit.ca/its/software/**

Any questions?

C++ PROGRAM STRUCTURE

C++

- Multi-paradigm
  - **Procedural**
  - **Object oriented**
  - Generic programming
- **Compiled** language
- Familiar type system (int, float, etc)
- Somewhat verbose
- Widely used
- Pass by value or pass by reference

Hello World

```
#include <iostream>

int main( )
{
     std::cout << "Hello world!" << std::endl;
     return 0;
}
```

<iostream>

```
#include <iostream>

int main( )
{
        std::cout << "Hello world!" << std::endl;
        return 0;
}
```

- This is a header file
- Note we wrap it in angle brackets and there is no file extension

- Java has an API
- C++ has a standard library*

- The <iostream> header contains some standard stream objects like **cout**

* http://en.cppreference.com/w/cpp/header

Can we access C header files in C++?

Yes!

They are included as a subset of the C++ standard library.

The names are a bit different:

math**.h** becomes <**c**math>

limits**.h** becomes <**c**limits>

stdlib**.h** becomes <**c**stdlib>

…and so on…

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

The insertion operator <<

- An overloaded function (we will learn how to do this later)
- We apply it to an output stream like cout
- Can be manipulated to format the output
  - Easier than printf in C
  - Much easier than NumberFormat, DecimalFormat, etc., in Java
  - We will explore manipulators in detail later

The scope operator ::

- Similar to the dot operator in C and Java

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

Aside: :: vs .

Q: When do we use the scope resolution operator :: ?

A: To access **members of a namespace or class**

Q: When do we use the dot operator . ?

A: To access **members of an object** (an instance of a class)

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

std::cout

- Predefined object of type ostream in the standard C++ library
- aka the **standard output stream** (stdout in C, System.out in Java)

```
int n = 12;
cout << n;
```

This is what actually happens:

```
cout.operator<<(n);*
```

* The function header looks like this: ostream& operator<<(int);

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

std::endl

- Called an '**output manipulator**' (we will examine manipulators later this term)
- Inserts a new-line and flushes the stream
- IO Stream objects in C++ (cin, cout) use an internal buffer of type streambuf
- Sometimes not necessary (we can just append \n to our output)

The main method

Everything starts with the main method (just like Java and C)

```
int main()
int main(int argc, char ** argv)
```

The main method returns an int (0 by default)

- Main method returning 0 indicates code ran without errors
- Best practice to leave `return 0;` at end of main method.
- However, code will still run if return is omitted

Preprocessor directives

```
#include <iostream>

int main( )
{
        std::cout << "Hello world!" << std::endl;
        return 0;
}
```

- **Instructions for the preprocessor, not the compiler**
- **NOT followed by a semi-colon (ends with new line)**
- Can use to include the header file for a library (#include)
- Can use to define constants (#define)
- Can use for conditional compilation (#ifdef, #ifndef, #if, #endif, etc.)

Namespaces

- Similar to a Java package
- **Prevents name collisions**
- Functions and objects defined in the standard C++ library are in the **std** namespace

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

The **using** keyword

- Just like Java's import
- Saves typing
- We can write **using namespace std**
- If we do this, we can write
  - cout instead of std::cout
  - endl instead of std::endl

```
using namespace std;
std::cout << "Hello world!" << std::endl;
```

Namespace and using option 1

```
#include <iostream>

//ALL names from std namespace visible in this FILE
using namespace std;

int main( )
{
     cout << "Hello world!" << endl;
     return 0;
}
```

Namespace and using option 2

```
#include <iostream>

//ONLY cout and endl from std namespace visible in this
FILE
using std::cout;
using std::endl;

int main( )
{
     cout << "Hello world!" << endl;
     return 0;
}
```

Namespace and using option 3

```
#include <iostream>

int main( )
{
     //ONLY cout and endl from std namespace visible
     in this FUNCTION
     using std::cout;
     using std::endl;
     cout << "Hello world!" << endl;
     return 0;
}
```

LET'S TRY IT

- Show HelloWorld.cpp in CLion

Header and Source file structure

- Unlike Java, C++ separates code into two files known as header and source files
- Header files declare interfaces to functions, classes etc. Other files use these interfaces to interact with your code
- Source files implement the declarations made in the header file

Header and Source file structure

- Let's say I ask you to *create a simple calculator that has an add function. Eventually an external file will call this add function.*
- Start by creating a **header file (calculator.hpp)** that contains a *function declaration/signature* for the add function
- A ***function declaration/signature*** contains the function return type, name, parameters, but no body/implementation
- End with a semicolon ;

calculator.hpp
int add(int first, int second);

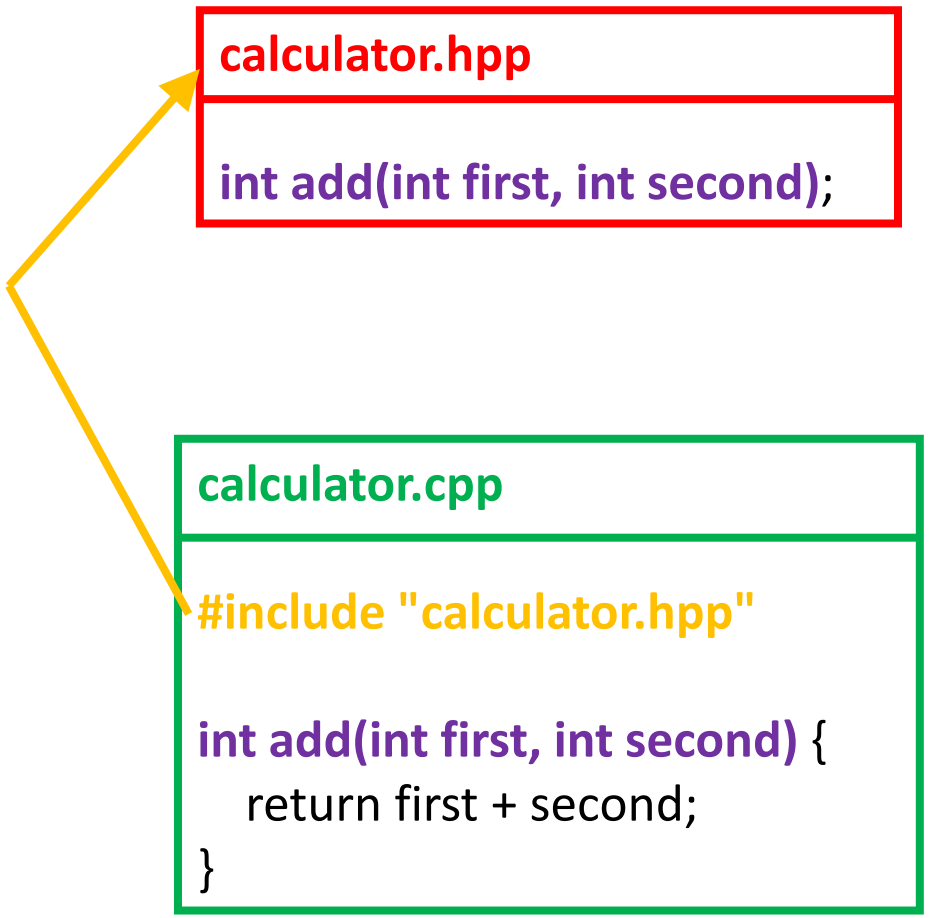
Header and Source file structure

- Next create the associated **source file (calculator.cpp)** that will implement the add function
- This file contains the function signature with the **full implementation** of the add method
- Pro-tip: the **function signature** should match the header's *function declaration/signature* for clarity

calculator.cpp

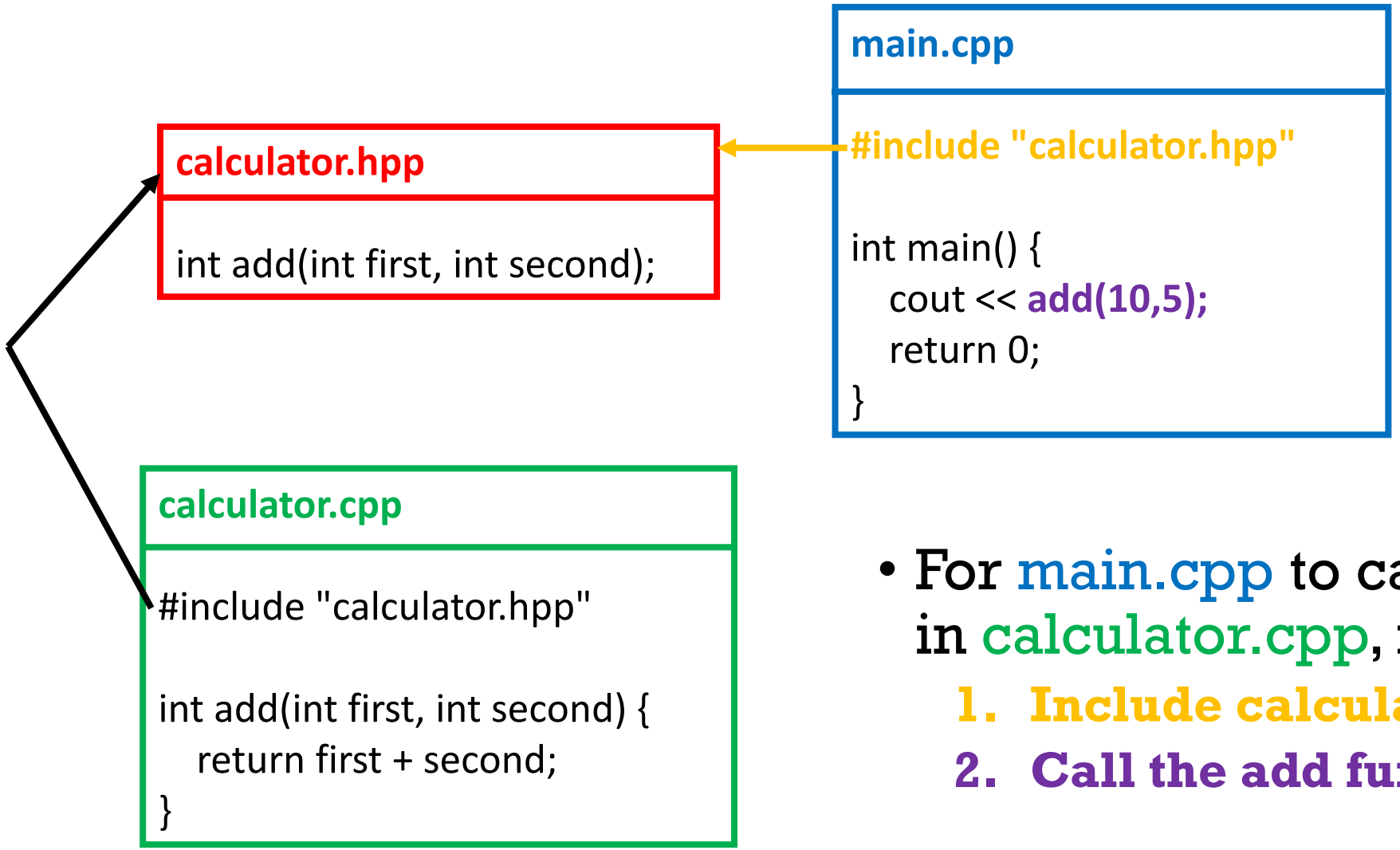
```
#include "calculator.hpp"

int add(int first, int second) {
    return first + second;
}
```

Header and Source file structure



- Make sure to "connect" the source file to the header file by having #include "calculator.hpp"
- Having the include statement and same **function signatures** in both files indicates the source file is implementing the header's declarations

Header and Source file structure



- For main.cpp to call the add function in calculator.cpp, it must:
  1. **Include calculator.hpp**
  2. **Call the add function**

Header and Source file structure

- In summary:
  - Source files should always be connected with a corresponding header file (except main.cpp)
  - Header files contain declarations intended to be used by other files
  - Source files implement code declared in header files
    - Source files can also include code not declared in header files
  - #include statements "connect" multiple files together. This allows external files to access functions, classes etc exposed in header files

HelloWorld.cpp

Some important notes about C++ programs:

1. One main method
2. main method may call other functions, just like C
3. In C++, the source file is a .cpp file
4. In C++, the header file is a .hpp file
5. Put function declarations in the header file
6. Put function signature and definitions in the source file
7. Use **#pragma once** instead of #ifndef to ensure header file is only included ONCE

FUNDAMENTAL TYPES

C++ is a strongly typed language

- Every variable has a type
- That type never changes
- Variable declarations need:
  1. Type
  2. Variable name
  3. Optional initialization

```
int num = 1;
```

Speaking of variables…

- ✓ Letters
- ✓ Digits (don’t start with digit)
- ✓ Underscores

- Begin with a letter or (**rarely**) an underscore

C++ common fundamental types

- bool – true/false. Represented as 1 or 0
- integer types
  - **char**
  - short
  - **int**
  - long
- floating point types
  - **float**
  - **double**
  - long double
- **void** – no type

Initializing variables in C++

There are 3 ways to initialize variables in C++:

1. C-like initialization

   int x = 0; // assignment operator
2. Constructor initialization (C++)

   int x(0); // parentheses
3. **Uniform initialization (C++11)**

   int x {0}; // curly braces

Why should we prefer uniform initialization?

It **prohibits implicit narrowing conversion** among built-in types

```
double x, y, z;
int sum = x + y + z; // ok (value of expression truncated to int)
int sum(x + y + z); // same

int sum{x + y + z}; // ERROR!  This won't work.  We're happy!
```

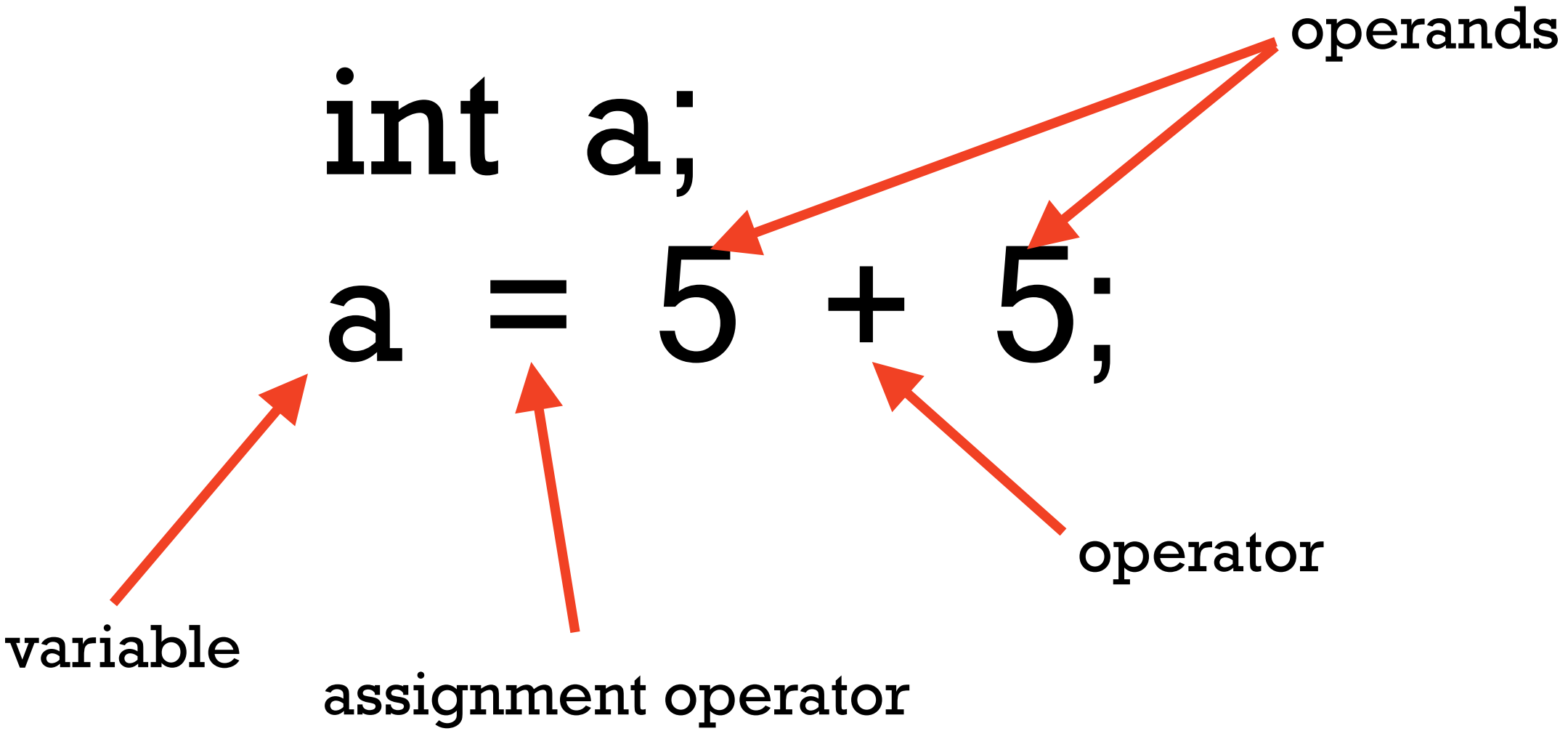
Agenda

1. Operators
2. C-style casting
3. Constants
4. Console IO

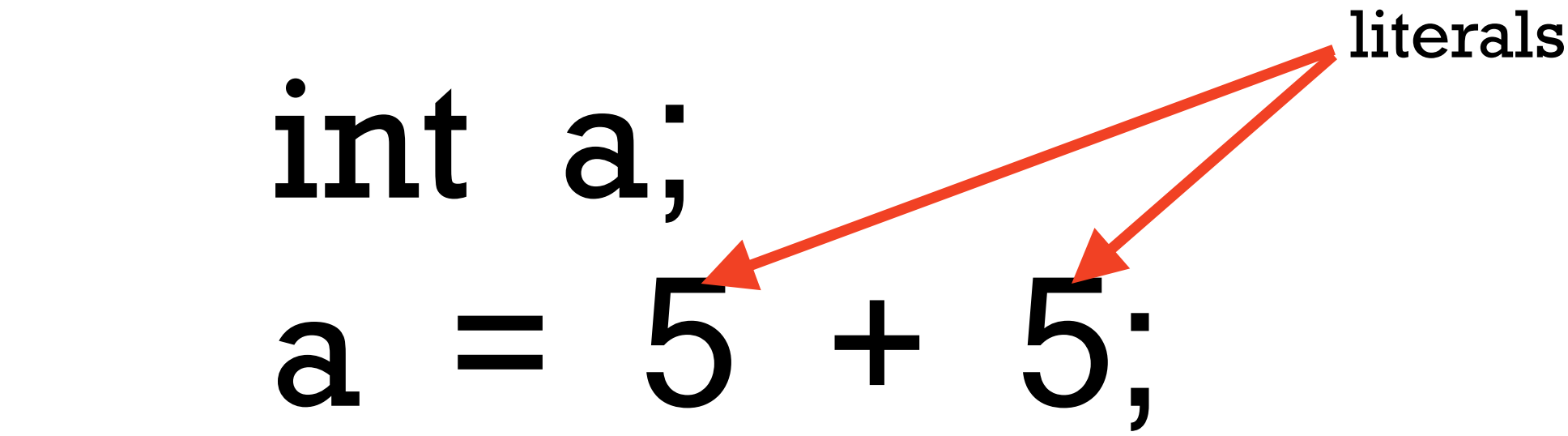
COMP 3522

OPERATORS

Operators and operands



Literals



Arithmetic operators

C++ has the usual set of arithmetic operators:

1. + Addition
2. - Subtraction
3. * Multiplication
4. / Division
5. % Modulo

Compound operators

These should all be familiar to you:

1. +=
2. -=
3. *=
4. /=
5. %=

Increment and decrement operators

1. ++
2. --

Remember pre vs post!

```
int x;
x++; //post
++x; //pre
```

Relational and comparison operators

1. ==
2. !=
3. >
4. >=
5. <
6. <=

Logical operators

1. !
2. &&
3. ||

Some more assorted operators

1. ? : (ternary operator)
2. , (comma operator, yuck)
3. ( ) (casting operator)
4. sizeof
5. And more (later this term)…

assortedOps.cpp

CASTING

Casting in C++

C-style casting (using parentheses) works in C++

```
float x = 2.25;
int y = (int)x;  //truncates value
```

(Later this term we will learn about C++ casting operators)

CONSTANTS

Constants in C++

What's a constant?

A value that can not change after it's initialized (Java *final*)

Old-style – a preprocessor directive:

#define PI 3.1415926535

(Remember: no semi-colon!)

Immutability with const

```
const int some_value = 1;
some_value = 9; // ERROR!
const int some_other_value; // ERROR!
const float pi(3.14159);
const char top_score{‘A’};
const bool larger{some_value < pi};
```

It is mandatory to set a const value in its declaration!

Useful as a modifier for function parameters.

```
void myFunc(const int num) {…}
```

“I promise not to change this value.”

Immutability with constexpr (more later!)

```
constexpr double another_value{1.3};
```

“To be evaluated at compile time.”

- Think of it as a compile-time constant
- Useful for performance

When to use const vs constexpr

- Will be important when we talk about:
  - Static variables
  - Constructors
- For now:
  - A constexpr **must be** assigned a value by compile time
  - A const **can be** assigned a value after compile time

Example

```
#include <iostream>
using namespace std;
int main()
{
	int input;
	cin >> input; // WOW WHAT'S THIS?
	const int constant_input = input;
	cout << constant_input << endl;
	return 0;
}
```

const_vs_constexpr.cpp

CONSOLE IO

Formatting output: member functions

- Recall std::**cout** is a **global object** of class **ostream**
- Recall in Java, behaviors are called methods
- In C++, we call them **member functions**
- Check out the member functions here: http://en.cppreference.com/w/cpp/io/basic_ostream

Member functions

What do these lines of code do?

```
cout.setf(ios_base::fmtflags);
cout.unsetf(ios_base::fmtflags);
```

The std::ios_base superclass of std::basic_ostream defines ios_base::fmtflags that we can use to **format output**:

http://en.cppreference.com/w/cpp/io/basic_ostream
http://en.cppreference.com/w/cpp/io/ios_base/fmtflags

Some rules

setf(flag) and unsetf(flag)

- Argument can be:
  - boolalpha
  - showbase
  - uppercase
  - showpos

setf(flag, flag)

- Arguments can be:
  - dec/oct/hex, basefield
  - fixed/scientific, floatfield
  - left/right/internal, adjustfield

Change bool display and print in hex explicitly

- Set booleans to show “false” or “true” instead of 0 or 1

```
cout.setf(ios_base::boolalpha)
```

- Printing in hex in C with printf requires a lot of typing
- Printing in hex in C++ is almost too easy:

```
int n{15};
cout.setf(ios_base::hex, ios_base::basefield);
cout << n << endl; // hex value - f
```

Less verbose: output manipulators

- Printing in hex in C with printf requires a lot of typing
- Printing in hex in C++ is almost too easy:

```
int n{15};
cout << hex << n << endl; // hex value - f
```

We call these **output manipulators**.

https://en.cppreference.com/w/cpp/io/manip

Under the hood

Behind the scenes, the **hex** output manipulator calls a **hex** function that calls **setf** with **hex**, and **basefield** flags

```
cout << hex << n << endl;
```

```
ostream& hex(ostream& outputstream)
{
    outputstream.setf(ios_base::hex,
                      ios_base::basefield);
    return outputstream;
}
```

Output manipulators

- **showpos/noshowpos** - assuming *n* is now 123

```
cout << showpos << n;   // +123
cout << noshowpos << n;      // 123
```

- **dec/hex/oct**

```
cout << dec << n; // 123
cout << hex << n; // 7b
cout << oct << n; // 173
```

Output manipulators

- **uppercase/nouppercase**

```
cout << uppercase << hex << n;  // 7B
cout << nouppercase << hex << n; // 7b
```

- **showbase/noshowbase**

```
cout << showbase << hex << n << endl; // 0x7b
cout << noshowbase << hex << n << endl; // 7b
```

Output manipulators

- **left/internal/right** - assuming *n* is -123

```
cout << setw(6) << left << n;     // |-123   |
cout << setw(6) << internal << n; // |-  123|
cout << setw(6) << right << n;    // |  -123|
```

- **showpoint/noshowpoint** - assuming d1 = 100.0 and d2 == 100.12

```
cout << noshowpoint << d1 << " " << d2; // 100 100.12
cout << showpoint << d1 << " " << d2;   // 100.000 100.120
```

Output manipulators

- **fixed/scientific** - assuming *number* is 123.456789

```
cout << fixed << number;        // 123.456789
cout << scientific << number;   // 1.234568E+02
```

- **boolalpha/noboolalpha** - assuming *fun* is true

```
cout << boolalpha << fun;       // true
cout << noboolalpha << fun;     // 1
```

Output manipulators with <iomanip>

- **setw(value)** sets minimum width for one field only

```
cout << setw(5) << number; // |  123| if n = 123
```

- **setfill(fillchar)**

```
cout << setfill('*') << setw(5) << number; // prints **123
```

Output manipulators with <iomanip>

- **setprecision(value)**

```
// assuming number is 123.4567845678
cout << setprecision(7) << number; // 123.4568
streamsize prec = cout.precision();
```

Note: default precision = 6

Member functions vs output manipulators

Member Function	Output Manipulator
`cout.setf(ios_base::showpos);` `cout << number;`	`cout << showpos << number;`
`cout.width(5);` `cout << number;`	`cout << setw(5) << number;`

Which looks easier?

What about **input? Extraction operator >>**

- Getting input with Java requires a scanner and a non-trivial amount of code
- Getting input with C is dangerous and requires finesse with fgets and sscanf (recall scanf was not our friend)

- C++: use std::cin

```
int m, n;
cin >> m >> n; // Input 12 34, or 12 <enter> 34
```

precisionAndWidth.cpp

Read an int

```
int hours;
cin >> hours;
cout << “Today I slept for “ << hours
     << “hours” << endl;
```

Read a floating point number

```
double weight_kg;
cin >> weight_kg;
cout << “I weigh “ << weight_kg
     << “kilos” << endl;
```

It's not infallible, though!

```
constexpr int first_name_length = 5;
char first_name[first_name_length];
cin >> first_name; // NOOOOOO DON'T DO THIS
```

Recall that char[] == char *

cin doesn't know the length of the array

We have a memory allocation issue

But we can fix it!

```
#include <iomanip>

constexpr first_name_length = 5;
char first_name[first_name_length];
cin >> setw(5) >> first_name;
```

readName.cpp

IO: input I

- What if input fails?
- ios_base::iostate contains:
  - ios_base::**failbit** (operation failed)
  - ios_base::**badbit** (stream error)
  - ios_base::**eofbit** (set on EOF)
  - ios_base::**goodbit** (zero – no bits sets)
- cin is true if cin.fail() is false:

```
int n;
if (cin >> n)…
```

https://en.cppreference.com/w/cpp/io/ios_base/iostate

IO: input II

- You can test these bits with cin's member functions:
  1. **fail()** – true iff badbit or failbit are set
  2. **bad()** – true iff badbit is set
  3. **eof()** – true iff eofbit is set
  4. **good()** – true iff goodbit is set (no bits are set)

Hint: call cin.clear() after an input failure!

IO: Input Examples

```
int n;
cin >> n
//Assume * represents the EOF
```

User Input	n	failbit	eofbit
123 456	123	Not set	Not set
123*	123	Not set	Set
hello	No change	set	Not set
*	No change	set	set

IO: Ignoring input

- Recall cin is an istream
- std::basic_stream has a member function called **ignore**
- If cin fails, use **ignore** to clear bad data in its buffer

```
ignore()    // skips/ignores/tosses 1 char
ignore(128) // skips 128 char or until EOF
ignore(128, ‘\n’) // skips 128 until EOF or ‘\n’
ignore(LLONG_MAX, ’\n’) // Throws away LLONG_MAX
                        // char
```

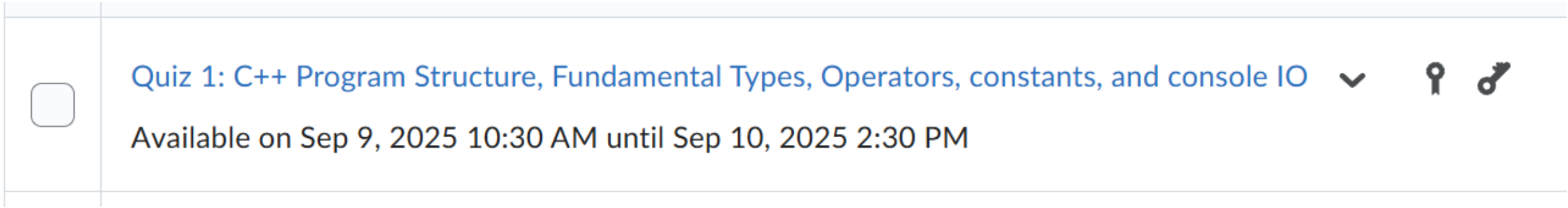
IO: Throwing away an entire line

```
#include <limits>

cin.clear(); //unsets failbits
cin.ignore(numeric_limits<streamsize>::max(), '\n');

//custom helper method to quickly clean cin
void ignoreline(istream& is)
{
	is.clear(); //unsets failbits
	is.ignore(numeric_limits<streamsize>::max(), '\n');
}
```

AddIntegersError.cpp
AddIntegers.cpp

Quiz next week during lab

- Covers week 1 material



- Learning hub quiz
  - Multiple choice
  - True/False
  - Matching options