Class 03

Operators, For-Loops, Functions
Introduction to Simulations

Outline

- Operators
- For Loops
- Functions
- Introduction to Simulations

Operators

- We have already seen some basic operators (=, +, -, *, /, <<, >>) as well as some comparison operators (==, !=, <, >, <=, >=).
- Operators take one or more pieces of data and transform them, either producing a new value and leaving the original(s) intact or mutating the original (dropping the old value).
  - e.g. `auto a = b + c;`
- There are many more operators in C++! Here are a few more we want to learn about:
  - `%`
  - `++, --`
  - `+=, -=, *=, /=, %=`

Operators - Division

- Division is usually straightforward, but there is something we need to be aware of: there are two kinds of division!
  - division and integer division
- The first division is the division that we know and love.
  - e.g. 5 / 2 is 2.5
  - e.g. 100 / 25 is 4
- *Integer division* is the division that is used when we are working with integers in C++. This is the division that we learn in grade school before we learn about decimal values.
  - e.g. 10 / 3 = 3
  - e.g. 50 / 17 = 2

Operators - Modulus

- % is called the modulus operator. This takes two numbers and returns the remainder after performing integer division.
- Modulus only works with integers, so you cannot use this operator with floating point data.

```
auto a = int{20 % 7};        // here a is assigned the value 6
auto b = int{8 % 3};         // here b is assigned the value 2
auto c = double{3.314 % 3};  // ERROR! cannot compute modulus of floats/doubles
```

Operators - Increment

- ++ and -- are called the increment operators. These operators work specifically on *variables* and respectively increase the value of the variable by 1 or decrease the value of the variable by 1.

- Again, these only work on variables! This is because they *modify the source data*.

```
auto a = int{10};
a++;   // a is now 11!
10++;  // ERROR! 10 is not a variable, and so we cannot ++ it!
```

Operators – Assignment

- We have already seen the basic assignment operator (=), but there are a handful of others that work a little differently.

- +=, -=, *=, /=, %= are also assignment operators. They assign a new value to an existing variable, performing an additional operation with the input.

- These also **<u>only work</u>** on variables! This is because they *modify the source data*. The operator "before" the = tells us how the data is modified.

```
auto a = int{10};
a *= 123;     // a is now 1230!
10 *= 123;    // ERROR! 10 is not a variable, and so we cannot *= it!
```

Improper use of operators

- What is wrong here?

```
#include <iostream>
using namespace std;
auto main() -> int
{
    auto a = int{10};
    auto b = int{100};
    a + b;
}
```

Improper use of operators

- What is wrong here?

```
#include <iostream>
using namespace std;
auto main() -> int
{
    auto a = int{10};
    auto b = int{100};
    a + b;
}
```

- The code is valid, but the operation performed on a and b is lost, because we never do anything with the new value it produced!

Improper use of operators

- What is wrong here?

```cpp
#include <iostream>
using namespace std;
auto main() -> int
{
    auto a = int{1 += 2};
}
```

Improper use of operators

- What is wrong here?

```cpp
#include <iostream>
using namespace std;
auto main() -> int
{
    auto a = int{1 += 2};
}
```

- You cannot use += without a variable! Specifically, the left-hand side of the += needs to be a variable, since += wants to modify the source of the data!

Improper use of operators

- What is wrong here?

```
#include <iostream>
using namespace std;
auto main() -> int
{
    auto a = double{19 / 3};
    cout << a << endl;
}
```

Improper use of operators

- What is wrong here?

```
#include <iostream>
using namespace std;
auto main() -> int
{
    auto a = double{19 / 3};
    cout << a << endl;
}
```

- This is valid C++, but what is the value of a? It is a double, but its value is just 6. This is because the division is integer division, C++ does not care that you are assigning that value to a double.

Improper use of operators

- What is wrong here?

```
#include <iostream>
using namespace std;
auto main() -> int
{
    auto a = double{19.0 / 3.0};
    cout << a << endl;
}
```

- This is valid C++, but what is the value of a? It is a double, but its value is just 6. This is because the division is integer division, C++ does not care that you are assigning that value to a double.

  If at least one operand is a floating point it will be considered as standard division.

For Loops

- A common pattern for a while loop is to use some counter; give the counter an initial value, and while that counter is less than some limit, perform some actions and then increment the counter.

```
auto i = int{1};
auto sum = int{0};
while (i < 101)
{
    sum += i;
    i++;
}
cout << sum << endl;
```

- Here, i is our counter, starting at 1 and incremented every iteration by 1 until 101. During each iteration, we add i to the sum. Once the loop is over, we print the sum.

For Loops

- A common pattern for a while loop is to use some counter; give the counter an initial value, and while that counter is less than some limit, perform some actions and then increment the counter.

```
auto i = int{1};
auto sum = int{0};
while (i < 101)
{
    sum += i;
    i += 1;
}
cout << sum << endl;
```

```
auto sum = int{0};
for (auto i = int{1}; i < 101; ++i)
{
    sum += i;
}
cout << sum << endl;
```

- Here, i is our counter, starting at 1 and incremented every iteration by 1 until 101. During each iteration, we add i to the sum. Once the loop is over, we print the sum.

For Loops

- For loops use the following structure:

```
for (initial; condition; post-action)
{
    action(s);
}
```

- We will see later just how useful this structure is for expressing certain algorithms.

- Do note though that for loops are just specialized while loops, and so anything you can express one you can do so with the other.

For Loops

- Example

Starting with i = 0, if i < 10, do something,
then increment i by 1 and repeat

```
auto a = int{0};
auto b = int{1};
for (auto i = int{0}; i < 10; ++i)
{
    auto c = a + b;
    a = b;
    b = c;
    cout << c << endl;
}
```

- What is this doing? What is the output?

For Loops

- Example

```
auto a = int{0};
auto b = int{1};
for (auto i = int{0}; i < 10; ++i)
{
    auto c = a + b; // note we are omitting the {}; this is ok
    a = b;
    b = c;
    cout << c << endl;
}
```

- What is this doing? What is the output?

For Loops

- Example

```
auto a = int{0};
auto b = int{1};
for (auto i = int{0}; i < 10; ++i)
{
    auto c = a + b;
    a = b;
    b = c;
    cout << c << endl;
}
```

- This is generating the first 10 elements of Fibonacci! While the 0 and 1 are not printed, we do produce the following sequence:

1

2

3

5

8

13

21

34

55

89

For Loops

- Example

```
auto data = vector<int>{1, 2, 3, 4, 5, 6, 7, 8, 9};
for (auto pos = size_t{0}; pos < data.size(); ++pos)
{
    cout << data[pos] << endl;
}
```

- Let's assume **data** is a list of numbers, and we can access elements of the list using their position of the list. *size_t* is a special integer type for representing the length of the list.

- What is this doing? What is the output?

Improper Use of For Loops

- What's wrong here?

```
#include <iostream>
using namespace std;
auto main() -> int
{
    for (a = 0; a < 10; ++a)
    {
        cout << a << endl;
    }
}
```

Improper Use of For Loops

- What's wrong here?

```
#include <iostream>
using namespace std;
auto main() -> int
{
    for (a = 0; a < 10; ++a)
    {
        cout << a << endl;
    }
}
```

- We are not declaring the type of a! This is incredibly odd looking as is...

Improper Use of For Loops

- What's wrong here?

```
#include <iostream>
using namespace std;
auto main() -> int
{
    for (auto a = int{0}; a < 10; ++a)
    {
        cout << a << endl;
    }
}
```

- This is better!

Improper Use of For Loops

- What's wrong here? We defined the type of a!

```
#include <iostream>
using namespace std;
auto main() -> int
{
    for (auto a = int{100}; a < 10; ++a)
    {
        cout << a << endl;
    }
}
```

Improper Use of For Loops

- What's wrong here? We defined the type of a!

```
#include <iostream>
using namespace std;
auto main() -> int
{
    for (auto a = int{100}; a < 10; ++a)
    {
        cout << a << endl;
    }
}
```

- We will never enter this loop, because a is already greater than 10!

Improper Use of For Loops

- What's wrong here? We defined the type of a and we ***will*** enter the loop!

```
#include <iostream>
using namespace std;
auto main() -> int
{
    for (auto a = int{100}; a < 10; --a)
    {
        cout << a << endl;
    }
}
```

Improper Use of For Loops

- What's wrong here? We defined the type of a and we ***will*** enter the loop!

```
#include <iostream>
using namespace std;
auto main() -> int
{
    for (auto a = int{100}; a < 10; --a)
    {
        cout << a << endl;
    }
}
```

- This is an infinite loop, as a will always be less than 10!

Improper Use of For Loops

- What's wrong here?

```
#include <iostream>
using namespace std;
int main()
{
    int a;
    for (a = 0; a < 10; ++a)
    {
        cout << a << endl;
    }
}
```

Improper Use of For Loops

- What's wrong here?

```
#include <iostream>
using namespace std;
int main()
{
    int a;
    for (a = 0; a < 10; ++a)
    {
        cout << a << endl;
    }
}
```

- This is valid code but avoid doing this in C++. Those with a C background do this *all the time*, and it is not *good C++*. Not only is a being declared without an initial value, but there is no reason to not declare it within the for-loop.

Functions

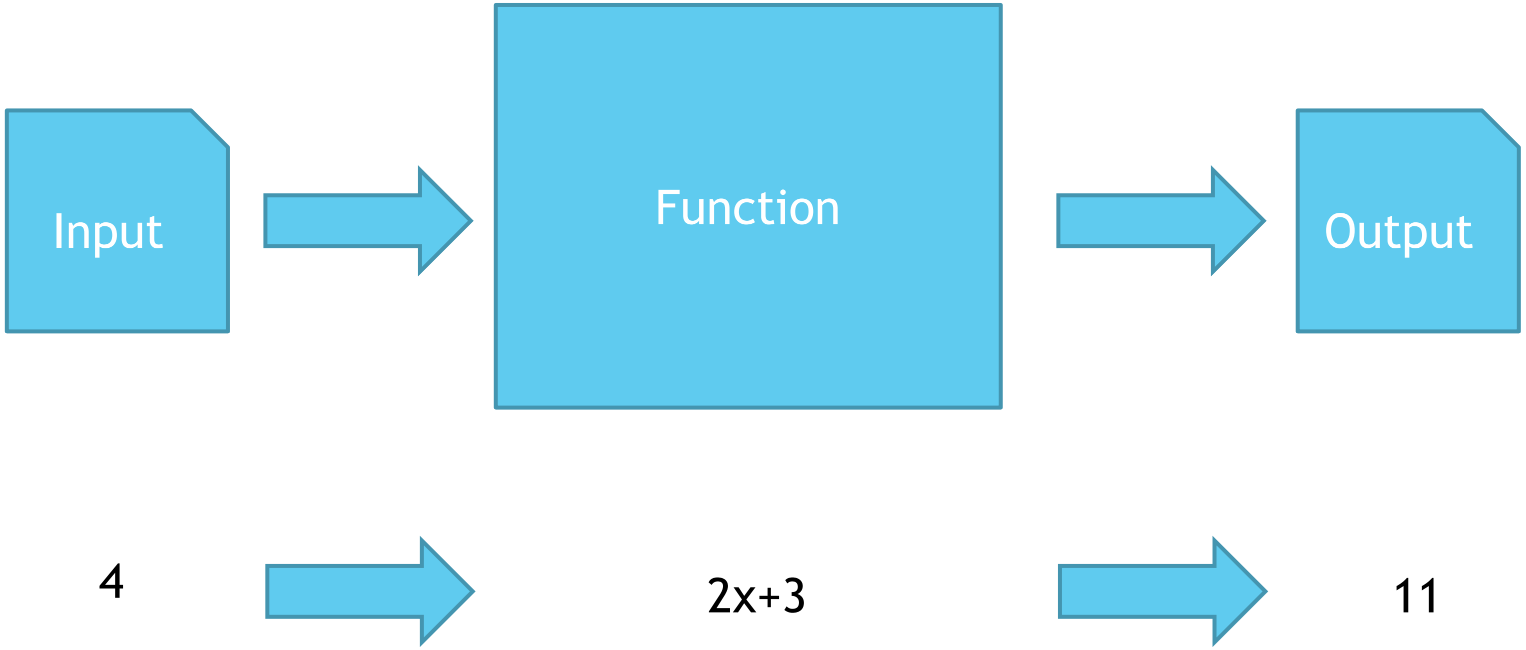
- Functions are *callable* pieces of code that perform some set of actions.

- Functions in C++ are very analogous to functions in mathematics. Consider:

  $f(x) = 2x + 3$

  This function takes in some input x, performs a multiplication and addition, and then returns a new number. E.g. $f(4) = 2 * 4 + 3 = 11$

Functions



Functions

- We use functions to encapsulate and separate commonly executed code.
- Functions can perform computations, log data, and/or anything we need it to do.
- Functions in C++ differ from mathematical functions in that mathematical functions always have inputs and outputs, whereas in C++ they are optional.
- When functions *return* something, we need to explicitly do something with that data! Otherwise it will just get tossed out by the computer.

Functions

- Functions take the following form:

```
auto function_name(inputs) -> return_type
{
    action(s);      // optional
    return output;  // optional
}
```

- function_name: name of the function, can by anything
- return_type: the type of data the function returns, e.g. int, double, void
- inputs: variables (with their type) input to the function, if any
- action(s): any number of actions to take
- output: the data to return from the function, if any.

Functions

- Example

```
auto f(double x) -> double
{
    return 2.0 * x + 3.0; // 2x+3
}
```

Functions

- Example

```
auto compute_factorial(int x) -> int
{
    auto f = int{1};
    for (auto i = int{2}; i <= x; ++i)
    {
        f *= i;
    }
    return f;
}
```

Functions

- Example

```
#include <iostream>
#include <numbers>
using namespace std;
using namespace std::numbers;

auto compute_sphere_surface_area(double radius) -> double
{
    return 4.0 * pi * radius * radius;
}

auto main() -> int
{
    compute_sphere_surface_area(1.0);
}
```

Functions

- Example

```
#include <iostream>
#include <numbers>
using namespace std;
using namespace std::numbers;

auto compute_sphere_surface_area(double radius) -> double
{
    return 4.0 * pi * radius * radius;
}

auto main() -> int
{
    auto sa = compute_sphere_surface_area(1.0);  // omit the {} and deduce type
    cout << sa << endl;
}
```

Functions

- Example

```
#include <iostream>
#include <numbers>
using namespace std;
using namespace std::numbers;

auto compute_sphere_surface_area(double radius) -> double
{
    return 4.0 * pi * radius * radius;
}

auto main() -> int
{
    for (auto radius = double{1.0}; radius <= 100.0; radius *= 10.0)
    {
        auto sa = compute_sphere_surface_area(radius);
        cout << sa << endl;
    }
}
```

Introduction to Simulations

- Simulations are replications of (sometimes real) **systems**.
- Systems in our reality can be described by mathematical formula.
- Simulations have *entities*, or objects, within the simulated world that experience the simulated world, affect the simulated world, or both.

Entities

- Entities have spatial components that align them with the space of the simulated world.
- Entities have temporal components that align them with the time of the simulated world.
- Entities also have behavioral components that dictate how they experience and affect the simulated world.
- The *state of an entity* aligns their spatial and behavioral components with the temporal component. As time evolves, so does the state.

Spatial Components

- Spatial components are components that exist within a space, following the properties and rules of that space.
  - Coordinate planes, grids, hexgrids, etc.
  - Euclidean space with Euclidean distance
- We will simulate components within 2D (and 3D space later, too).
- These components will have positions, and later this semester shape and orientation.

Temporal Components

- Temporal components are components that experience *time*.

- We will simulate basic notions of time by "keeping time" within our programs.

- This usually takes the form of a single variable that we increment repeatedly. A side effect of trying to represent temporal components (which is largely unavoidable no matter what we try) is that we are explicitly working with discrete time, rather than continuous time.

Behavioral Components

- Behavioral components are components that define *how* an entity evolves within the simulated world.
  - e.g. an entity experiencing gravity
    - This affects the entity's spatial components, always pulling it downward
  - e.g. an entity passing on a disease to another entity
    - This affects the (other) entity's behavioral state, it can also now spread disease
- Many complexity problems in simulations arise here.
  - e.g. Consider a biologically and physically accurate simulation of an ant colony. We would need to simulate (not limited to):
    - biological systems of the ants (e.g. pheromones, reproduction)
    - colony representation (e.g. demographics)
    - physical constraints (e.g. collision detection)
    - environmental effects (e.g. weather conditions, soil conditions, food security, predators)
    - Hundreds of thousands/millions of ants per colony!
- We will simulate basic behaviors and systems, often oversimplifying them to get started.

Observations

- While we experience reality (time) continuously, it is not possible for us to simulate time in the same way.
- Via our software we will present time as a single (floating-point) number that we can increment to advance it.
  - E.g. we can start at time=0.0, and repeatedly increment it by 0.1s. As time advances, we instruct our entities and world to move forward in time **by the same increment**.
- This means that our simulations are going to be *discrete-time simulations*.
- Whenever we advance the time of our simulations, we will make *observations*.

Observations

- An *observation* is a *recording* of the state of the simulation at a specific time.
  - Where are all the entities? What are they currently doing? What does the world look like?
- Obviously, there are infinite observations we can make with arbitrarily small increments of time.
  - How many values of time are between 0.0 and 1.0? Infinite! (well maybe ~5.4×10^{44} if you are Max Planck)
- Continuous time looks like:

- Discrete time looks like:

- Note the gaps in our timeline – this is lost information!

Observations

- The more observations we make, the more CPU and memory (and maybe disk space!) are needed to make them.
  - Recording a single observation of 4 doubles, logged with a precision of 8 decimals and 4 leading characters requires ~13 bytes.
  - Recording the same four doubles over 1 million observation requires 13 million bytes, or 13Mb.
    - This could be a 1.0s simulation with a time delta of 0.000001
    - This could be a 1000.0s simulation with a time delta of 0.001
- More observations means having more information about the system, but there are **diminishing returns**.
  - Making more observations is only useful if the simulation is of a high enough *fidelity*.

Observations - Example

Time (s)	ID	X (m)	Y (m)
0.0	1	1.0	0.0
0.0	2	0.0	1.0
0.1	1	1.0	1.0
0.1	2	-1.0	0.0
0.2	1	1.0	2.0
0.2	2	-2.0	-1.0
0.3	1	2.0	3.0
0.3	2	-3.0	-2.0