Review: Types and Values

CS 115

Dr. Joseph Eremondi, adapted from Dr. Shakil Khan, Dr. Philip Fong, and Dr. Howard Hamilton

Last updated: February 14, 2025

Leading Towards Polymorphism and Generics

Midterm Details

- **March 7** in-class

Midterm Details

- **March 7** in-class
- Covers all topics up-to and including constructors/overloading

Midterm Details

- **March 7** in-class
- Covers all topics up-to and including constructors/overloading
- May cover both conceptual and practical (code)

The Road Ahead

- Next unit has a significant increase in the complexity of what we’re learning

The Road Ahead

- Next unit has a significant increase in the complexity of what we're learning
    - Powerful but complex features of C++

The Road Ahead

- Next unit has a significant increase in the complexity of what we're learning
    - Powerful but complex features of C++
- To lead to that, we're going to review some more of the basics

The Road Ahead

- Next unit has a significant increase in the complexity of what we're learning
    - Powerful but complex features of C++
- To lead to that, we're going to review some more of the basics
    - Help catch up

What is a value?

- The result of running some computation

What is a value?

- The result of running some computation
  - E.g. `2`, `true`, `{1,2,3,4}`, `3.1415926535`

What is a value?

- The result of running some computation
    - E.g. `2`, `true`, `{1,2,3,4}`, `3.1415926535`
- An *expression* has a value

What is a value?

- The result of running some computation
    - E.g. `2`, `true`, `{1,2,3,4}`, `3.1415926535`
- An *expression* has a value
    - e.g. `2+2` and `1+3` are different expressions that produce the same value

Things that have values

- Variables have values while we’re running our program

Things that have values

- Variables have values while we're running our program
    - But we might not know what that value is ahead of time

Things that have values

- Variables have values while we're running our program
    - But we might not know what that value is ahead of time
- Arithmetic expressions/using operators `3+4`, `x/9`, `2*3*4*5*6`

Things that have values

- Variables have values while we're running our program
  - But we might not know what that value is ahead of time
- Arithmetic expressions/using operators `3+4`, `x/9`, `2*3*4*5*6`
- Function calls

Things that have values

- Variables have values while we're running our program
  - But we might not know what that value is ahead of time
- Arithmetic expressions/using operators `3+4`, `x/9`, `2*3*4*5*6`
- Function calls
  - `f(x)`, `sqrt(2)`, `Counter(2,10)`

Things that have values

- Variables have values while we're running our program
    - But we might not know what that value is ahead of time
- Arithmetic expressions/using operators `3+4`, `x/9`, `2*3*4*5*6`
- Function calls
    - `f(x)`, `sqrt(2)`, `Counter(2,10)`
    - Exception: `void` return type

Things that have values

- Variables have values while we're running our program
  - But we might not know what that value is ahead of time
- Arithmetic expressions/using operators `3+4`, `x/9`, `2*3*4*5*6`
- Function calls
  - `f(x)`, `sqrt(2)`, `Counter(2,10)`
  - Exception: `void` return type
- Array access

Things that have values

- Variables have values while we're running our program
  - But we might not know what that value is ahead of time
- Arithmetic expressions/using operators `3+4`, `x/9`, `2*3*4*5*6`
- Function calls
  - `f(x)`, `sqrt(2)`, `Counter(2,10)`
  - Exception: `void` return type
- Array access
  - `A[3] + 2`

Things that have values

- Variables have values while we're running our program
  - But we might not know what that value is ahead of time
- Arithmetic expressions/using operators `3+4`, `x/9`, `2*3*4*5*6`
- Function calls
  - `f(x)`, `sqrt(2)`, `Counter(2,10)`
  - Exception: `void` return type
- Array access
  - `A[3] + 2`
- Class/struct field access

Things that have values

- Variables have values while we're running our program
  - But we might not know what that value is ahead of time
- Arithmetic expressions/using operators `3+4`, `x/9`, `2*3*4*5*6`
- Function calls
  - `f(x)`, `sqrt(2)`, `Counter(2,10)`
  - Exception: `void` return type
- Array access
  - `A[3] + 2`
- Class/struct field access
  - `point.x * -1`

Weird things that have values

- The value of `x = y` is `y`'s value

Weird things that have values

- The value of `x = y` is `y`'s value
    - So can do `x = y = z` to set all to `z`

Weird things that have values

- The value of `x = y` is y's value
    - So can do `x = y = z` to set all to `z`
- The value of `stream << "hello"` is another stream

Weird things that have values

- The value of `x = y` is y's value
  - So can do `x = y = z` to set all to `z`
- The value of `stream << "hello"` is another stream
  - So can chain them together

Weird things that have values

- The value of `x = y` is y's value
  - So can do `x = y = z` to set all to `z`
- The value of `stream << "hello"` is another stream
  - So can chain them together
- The value of `i++` is `i`'s original value

Weird things that have values

- The value of `x = y` is y's value
  - So can do `x = y = z` to set all to `z`
- The value of `stream << "hello"` is another stream
  - So can chain them together
- The value of `i++` is `i`'s original value
  - But `++i` has the value of `i` *after* adding one

Where you can use values

- Right hand side of an assignment

Where you can use values

- Right hand side of an assignment
    - `x = f(3);`

Where you can use values

- Right hand side of an assignment
    - `x = f(3);`
    - `A[3] = foo.x`

Where you can use values

- Right hand side of an assignment
    - `x = f(3);`
    - `A[3] = foo.x`
- Argument to a function/operation

Where you can use values

- Right hand side of an assignment
  - `x = f(3);`
  - `A[3] = foo.x`
- Argument to a function/operation
  - `printEntry({1, 2, 3, 4})`

Where you can use values

- Right hand side of an assignment
    - `x = f(3);`
    - `A[3] = foo.x`
- Argument to a function/operation
    - `printEntry({1, 2, 3, 4})`
- `return` in a function definition

Where you can use values

- Right hand side of an assignment
    - `x = f(3);`
    - `A[3] = foo.x`
- Argument to a function/operation
    - `printEntry({1, 2, 3, 4})`
- `return` in a function definition
- If a value has a certain type, you can use it *anywhere* that is expecting that type

What is a type?

- A type is a way of classifying values

What is a type?

- A type is a way of classifying values
- Tells you

What is a type?

- A type is a way of classifying values
- Tells you
    - What operations you can perform on a value

What is a type?

- A type is a way of classifying values
- Tells you
    - What operations you can perform on a value
    - Which functions can accept that value as an argument

What is a type?

- A type is a way of classifying values
- Tells you
  - What operations you can perform on a value
  - Which functions can accept that value as an argument
  - What you can do with the return value of a function

Values and Types

- Every value in C++ has a type

Values and Types

- Every value in C++ has a type
    - Some values have more than one type

Values and Types

- Every value in C++ has a type
    - Some values have more than one type
        - e.g. `1` can be `int`, `unsigned int`, `double` etc.

Values and Types

- Every value in C++ has a type
    - Some values have more than one type
        - e.g. `1` can be `int`, `unsigned int`, `double` etc.
        - More examples after the break

Values are NOT types

- `struct`, `class`, and `enum` are all ways of *defining a new type*

Values are NOT types

- `struct`, `class`, and `enum` are all ways of *defining a new type*
- They don't create any values in your program

Values are NOT types

- `struct`, `class`, and `enum` are all ways of *defining a new type*
- They don't create any values in your program
  - They create a whole collection of values that you can use in your program

Example: Struct

Example: Struct

```
struct Student{
  string name;
  int studentNum;
  float average;
}
```

- This defines a new type named `Student`

Example: Struct

```
struct Student{
  string name;
  int studentNum;
  float average;
}
```

- This defines a new type named `Student`
    - Doesn’t create any specific values

Example: Struct

```
struct Student{
  string name;
  int studentNum;
  float average;
}
```

- This defines a new type named `Student`
    - Doesn’t create any specific values
    - Just tells us what they might look like

Example: Struct

```
struct Student{
  string name;
  int studentNum;
  float average;
}
```

- This defines a new type named `Student`
  - Doesn’t create any specific values
  - Just tells us what they might look like
- Each `Student` value contains a value for *each* field

Example: Struct

```
struct Student{
  string name;
  int studentNum;
  float average;
}
```

- This defines a new type named `Student`
    - Doesn't create any specific values
    - Just tells us what they might look like
- Each `Student` value contains a value for *each* field
- Example values:

Example: Struct

```
struct Student{
  string name;
  int studentNum;
  float average;
}
```

- This defines a new type named `Student`
    - Doesn't create any specific values
    - Just tells us what they might look like
- Each `Student` value contains a value for *each* field
- Example values:
    - `{"Alice", 1234, 75.2}`

Example: Struct

```
struct Student{
  string name;
  int studentNum;
  float average;
}
```

- This defines a new type named `Student`
  - Doesn't create any specific values
  - Just tells us what they might look like
- Each `Student` value contains a value for *each* field
- Example values:
  - `{"Alice", 1234, 75.2}`
  - `{"Bob", 5678, 68.99}`

Example: Struct

```
struct Student{
  string name;
  int studentNum;
  float average;
}
```

- This defines a new type named `Student`
  - Doesn't create any specific values
  - Just tells us what they might look like
- Each `Student` value contains a value for *each* field
- Example values:
  - `{"Alice", 1234, 75.2}`
  - `{"Bob", 5678, 68.99}`
  - `{"Eve", 2468, 92.45}`

Example: Struct

```
struct Student{
  string name;
  int studentNum;
  float average;
}
```

- This defines a new type named `Student`
  - Doesn't create any specific values
  - Just tells us what they might look like
- Each `Student` value contains a value for *each* field
- Example values:
  - `{"Alice", 1234, 75.2}`
  - `{"Bob", 5678, 68.99}`
  - `{"Eve", 2468, 92.45}`
- Each `Student` contains a string AND an int AND a float

Types are Interchangeable

- Any type can be used as:

Types are Interchangeable

- Any type can be used as:
  - The type of a variable

Types are Interchangeable

- Any type can be used as:
    - The type of a variable
    - The return type of a function

Types are Interchangeable

- Any type can be used as:
  - The type of a variable
  - The return type of a function
  - The type of a function argument

Types are Interchangeable

- Any type can be used as:
    - The type of a variable
    - The return type of a function
    - The type of a function argument
    - Type type of things inside an array

Types are Interchangeable

- Any type can be used as:
    - The type of a variable
    - The return type of a function
    - The type of a function argument
    - Type type of things inside an array
    - The type of a class/struct field

Example: Declaring Variables

- Every time we declare (create a new) variable, it has the form `TypeName variableName;`

Example: Declaring Variables

- Every time we declare (create a new) variable, it has the form `TypeName variableName;`
- E.g. `Counter c;`

Example: Declaring Variables

- Every time we declare (create a new) variable, it has the form `TypeName variableName;`
- E.g. `Counter c;`
    - `Counter` is the name of the type;

Example: Declaring Variables

- Every time we declare (create a new) variable, it has the form `TypeName variableName;`
- E.g. `Counter c;`
    - `Counter` is the name of the type;
    - `c` is a name that we pick for the variable to have

Example: Declaring Variables

- Every time we declare (create a new) variable, it has the form `TypeName variableName;`
- E.g. `Counter c;`
  - `Counter` is the name of the type;
  - `c` is a name that we pick for the variable to have
- Works with any types

Example: Declaring Variables

- Every time we declare (create a new) variable, it has the form `TypeName variableName;`
- E.g. `Counter c;`
  - `Counter` is the name of the type;
  - `c` is a name that we pick for the variable to have
- Works with any types
  - `int x;`

Example: Declaring Variables

- Every time we declare (create a new) variable, it has the form `TypeName variableName;`
- E.g. `Counter c;`
  - `Counter` is the name of the type;
  - `c` is a name that we pick for the variable to have
- Works with any types
  - `int x;`
  - `Quadrant q;`

Example: Declaring Variables

- Every time we declare (create a new) variable, it has the form `TypeName variableName;`
- E.g. `Counter c;`
  - `Counter` is the name of the type;
  - `c` is a name that we pick for the variable to have
- Works with any types
  - `int x;`
  - `Quadrant q;`
  - `Point2D p;`

Example: Declaring Variables

- Every time we declare (create a new) variable, it has the form `TypeName variableName;`
- E.g. `Counter c;`
    - `Counter` is the name of the type;
    - `c` is a name that we pick for the variable to have
- Works with any types
    - `int x;`
    - `Quadrant q;`
    - `Point2D p;`
- May need constructor arguments, e.g. `Counter (3,4);`

Example: Declaring Variables

- Every time we declare (create a new) variable, it has the form `TypeName variableName;`
- E.g. `Counter c;`
  - `Counter` is the name of the type;
  - `c` is a name that we pick for the variable to have
- Works with any types
  - `int x;`
  - `Quadrant q;`
  - `Point2D p;`
- May need constructor arguments, e.g. `Counter (3,4);`
- C++ (usually) requires that we specify the type of every variable

Example: Enums

Example: Enums

```
enum Direction{
North, South, East, West};
```

- `Direction` is a type

Example: Enums

```
enum Direction{
North, South, East, West};
```

- `Direction` is a type
- There are exactly 4 values with type `Direction`

Example: Enums

```
enum Direction{
North, South, East, West};
```

- `Direction` is a type
- There are exactly 4 values with type `Direction`
    - `North`, `South`, `East`, and `West`

Types are Static

- **Static**: defined/checked at compile-time

Types are Static

- **Static**: defined/checked at compile-time
- **Dynamic**: defined/checked at run-time

Types are Static

- **Static**: defined/checked at compile-time
- **Dynamic**: defined/checked at run-time
- C++ Types are static

Types are Static

- **Static**: defined/checked at compile-time
- **Dynamic**: defined/checked at run-time
- C++ Types are static
  - Defined at compile-time

Types are Static

- **Static**: defined/checked at compile-time
- **Dynamic**: defined/checked at run-time
- C++ Types are static
  - Defined at compile-time
- Values are dynamic

Types are Static

- **Static**: defined/checked at compile-time
- **Dynamic**: defined/checked at run-time
- C++ Types are static
  - Defined at compile-time
- Values are dynamic
  - They exist in memory at run-time

Types are Static

- **Static**: defined/checked at compile-time
- **Dynamic**: defined/checked at run-time
- C++ Types are static
    - Defined at compile-time
- Values are dynamic
    - They exist in memory at run-time
    - Many values of a particular type may exist throughout the run of a program

Types in OOP

- A *class* is a special kind of type

Types in OOP

- A *class* is a special kind of type
    - Like a struct: specific fields with values

Types in OOP

- A *class* is a special kind of type
    - Like a struct: specific fields with values
- An *object* is a value of some class

Types in OOP

- A *class* is a special kind of type
    - Like a struct: specific fields with values
- An *object* is a value of some class
- Additionally: class has *methods*

Types in OOP

- A *class* is a special kind of type
    - Like a struct: specific fields with values
- An *object* is a value of some class
- Additionally: class has *methods*
    - Functions that are attached to a particular object

Types in OOP

- A *class* is a special kind of type
    - Like a struct: specific fields with values
- An *object* is a value of some class
- Additionally: class has *methods*
    - Functions that are attached to a particular object
    - Call using field-access dot notation

Types in OOP

- A *class* is a special kind of type
  - Like a struct: specific fields with values
- An *object* is a value of some class
- Additionally: class has *methods*
  - Functions that are attached to a particular object
  - Call using field-access dot notation
    - `point.print()`

Types in OOP

- A *class* is a special kind of type
    - Like a struct: specific fields with values
- An *object* is a value of some class
- Additionally: class has *methods*
    - Functions that are attached to a particular object
    - Call using field-access dot notation
        - `point.print()`
- Methods can access fields of the object called on

Types in OOP

- A *class* is a special kind of type
    - Like a struct: specific fields with values
- An *object* is a value of some class
- Additionally: class has *methods*
    - Functions that are attached to a particular object
    - Call using field-access dot notation
        - `point.print()`
- Methods can access fields of the object called on
    - Even private

Constructors

- A constructor creates an object of a given class

Constructors

- A constructor creates an object of a given class
- There may be more than one constructor

Constructors

- A constructor creates an object of a given class
- There may be more than one constructor
  - e.g. They can be *overloaded*

Constructors

- A constructor creates an object of a given class
- There may be more than one constructor
  - e.g. They can be *overloaded*
- We *define* each constructor once

Constructors

- A constructor creates an object of a given class
- There may be more than one constructor
    - e.g. They can be *overloaded*
- We *define* each constructor once
    - In the class declaration

Constructors

- A constructor creates an object of a given class
- There may be more than one constructor
  - e.g. They can be *overloaded*
- We *define* each constructor once
  - In the class declaration
- A constructor is called each time we *create* a variable of

Methods

Methods

```
Counter c;
...
c.increment();
```

- For a call `c.increment()`, the program:

Methods

```
Counter c;
...
c.increment();
```

- For a call `c.increment()`, the program:
  - Looks up the type of `c`

Methods

```
Counter c;
...
c.increment();
```

- For a call `c.increment()`, the program:
    - Looks up the type of `c`
        - e.g. `Counter`

Methods

```
Counter c;
...
c.increment();
```

- For a call `c.increment()`, the program:
    - Looks up the type of `c`
        - e.g. `Counter`
    - Looks at the `Counter` class definition for a `Counter::increment()` method

Methods

```
Counter c;
...
c.increment();
```

- For a call `c.increment()`, the program:
  - Looks up the type of `c`
    - e.g. `Counter`
  - Looks at the `Counter` class definition for a `Counter::increment()` method
  - Calls that method as a function

What’s Different in OOP

- Without OOP, functions exist separately from values

What’s Different in OOP

- Without OOP, functions exist separately from values
    - Can only access its arguments and globals

What’s Different in OOP

- Without OOP, functions exist separately from values
    - Can only access its arguments and globals
- With OOP, a method belongs to a particular value

What’s Different in OOP

- Without OOP, functions exist separately from values
    - Can only access its arguments and globals
- With OOP, a method belongs to a particular value
    - Most operations act on at least one value

What’s Different in OOP

- Without OOP, functions exist separately from values
    - Can only access its arguments and globals
- With OOP, a method belongs to a particular value
    - Most operations act on at least one value
    - So we treat the operation as if it is a part of that value

What’s Different in OOP

- Without OOP, functions exist separately from values
    - Can only access its arguments and globals
- With OOP, a method belongs to a particular value
    - Most operations act on at least one value
    - So we treat the operation as if it is a part of that value
- What this means?

What’s Different in OOP

- Without OOP, functions exist separately from values
    - Can only access its arguments and globals
- With OOP, a method belongs to a particular value
    - Most operations act on at least one value
    - So we treat the operation as if it is a part of that value
- What this means?
    - Different classes can have different methods of the same name

What’s Different in OOP

- Without OOP, functions exist separately from values
    - Can only access its arguments and globals
- With OOP, a method belongs to a particular value
    - Most operations act on at least one value
    - So we treat the operation as if it is a part of that value
- What this means?
    - Different classes can have different methods of the same name
    - When you call `foo.bar()`, it looks at the type of `foo` to know which code to use

Abstract Data Types

- A type where we *don't know exactly what values look like*

Abstract Data Types

- A type where we *don’t know exactly what values look like*
    - We just know what operations are supported for the type

Abstract Data Types

- A type where we *don’t know exactly what values look like*
    - We just know what operations are supported for the type
    - A collection of operations defining a specific *interface*

Abstract Data Types

- A type where we *don’t know exactly what values look like*
  - We just know what operations are supported for the type
  - A collection of operations defining a specific *interface*
  - Can only interact with the type through the interface

Abstract Data Types

- A type where we *don’t know exactly what values look like*
    - We just know what operations are supported for the type
    - A collection of operations defining a specific *interface*
    - Can only interact with the type through the interface
- Classes are the realization of the concept of an ADT

Abstract Data Types

- A type where we *don't know exactly what values look like*
    - We just know what operations are supported for the type
    - A collection of operations defining a specific *interface*
    - Can only interact with the type through the interface
- Classes are the realization of the concept of an ADT
    - The interface is the public methods of the class

Abstract Data Types

- A type where we *don't know exactly what values look like*
    - We just know what operations are supported for the type
    - A collection of operations defining a specific *interface*
    - Can only interact with the type through the interface
- Classes are the realization of the concept of an ADT
    - The interface is the public methods of the class
    - Can only interact with an object of a class through the interface