Chapter 1
Programming Foundations

1.1 Introduction

If you are reading this, there's a good chance you are preparing to take EECS 281 here at the University of Michigan. Some of you may have just completed EECS 280 and are looking forward to see what this next class offers. Others of you may have taken EECS 280 earlier and have not done any programming for over a semester. Regardless of which camp you are in, a lot of material covered in the class is brand new, so success in this class is fully attainable if you put in the time and effort.

That being said, EECS 281 is undoubtedly a challenging course. As an IA who taught this class for five semesters, I can certainly attest to the vastness and thought-provoking nature of the class material. However, this also makes EECS 281 quite fulfilling, as there is always something new to learn, regardless of where you are in your programming career. In fact, this was a motivating factor behind these notes in the first place; I wanted to consolidate everything I learned from three years of 281 experience into a single resource, so that students will be able to better understand and appreciate the full scope of the class material.

Before we begin, this first chapter will review some concepts from previous courses (mostly from EECS 280) that may be applicable to EECS 281. Thus, if you are already comfortable with any of this information, feel free to skip it. Throughout these notes, you may also see sections labeled with this asterisk symbol (✻). This indicates that the section contains bonus material that is not covered in the course, but rather something I personally think is useful or just good to know. This material usually includes C++ features that may be helpful for projects, algorithms and data structures that may show up during job interviews, as well as additional knowledge that may supplement material that is actually covered in the class. If you see an '✻' anywhere, feel free to skip its corresponding section (unless the material was actually explicitly covered in a lecture or lab, since it is possible for the course material to change over time). Similarly, if you see anything that you do not recognize in a section without an '✻' (and is not mentioned in the class at all), feel free to skip that material as well. ***The material covered in lectures and labs will determine what you are responsible for knowing in the class, and not the content in these notes.***

It should also be noted that this resource should NOT be used as a substitute for going to lab or lecture! Most of your learning will be done in the classroom, where you will have an opportunity to engage in an interactive study environment with your fellow classmates and instructors. However, feel free to use this text as a resource to supplement your studies. Good luck, and I hope you enjoy your experience in EECS 281!

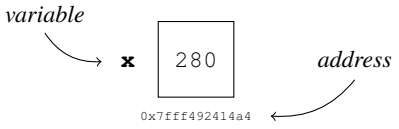
1.2 Pointers and References

※ 1.2.1 *Program Execution and the Machine Model*

To start off, we will begin by talking about the C++ *machine model* discussed in EECS 280, which allows us to visualize how code behaves during runtime. Consider the following code, which initializes an integer, copies that integer, and then assigns it to a new value:

```
1  int main() {
2    int x = 280;
3    int y = x;
4    x = 281;
5    y = x;
6  } // main()
```

The program's execution begins with `main()` on line 1. On line 2, we declare an integer `x` that is initialized with a value of `280`. When this line is executed during runtime, an integer object is created somewhere in memory. We denote this integer's memory location as its *address*.



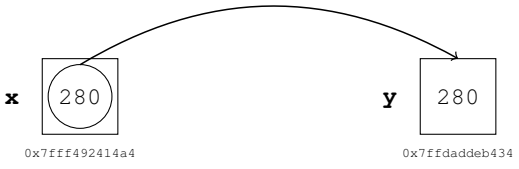
You do not need to worry about the specific memory address of the integer above (since it was arbitrarily chosen). In fact, when you write a program, you are generally *not* in control of the specific address locations that your objects are constructed in. Rather, the operating system decides the exact memory locations of each object during runtime.

Remark: If you want to get the address of an object in your program, you can use the ampersand symbol (`&`), which is known as the "address-of" operator. By placing an `&` next to an existing object in your program, you can identify its address location in memory. *Note that this usage should **not** be confused with the declaration of a reference, which also uses the ampersand symbol.*

```
1  int main() {
2    int x = 280;
3    std::cout << &x << std::endl; // sample output: 0x7fff492414a4
4  } // main()
```

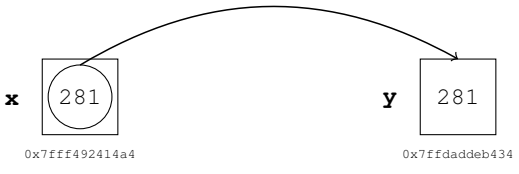
Addresses are typically denoted in *hexadecimal* (base-16) notation, with a leading `0x` followed by the digits `0-9` and the letters `a-f`, where `a` represents the value 10, `b` represents the value 11, `c` represents the value 12, `d` represents the value 13, `e` represents the value 14, and `f` represents the value 15. The size of an address depends on the system you are using. If you are using a 64-bit machine, each memory address takes up 64 bits (or 8 bytes). Alternatively, if you are using a 32-bit machine, each memory address takes up 32 bits (or 4 bytes).

On line 3, we initialize a variable `y` with the contents of `x`. Here, the program takes the value stored in the memory associated with `x` and copies it to the memory allocated for `y`. Note that, in C++, the assignment of `y = x` does *not* assign `y` to the same memory object referred to by `x`! Instead, a new object is created for `y` with the value of `x` copied over, and `x` and `y` refer to different objects in memory. This is a rather important distinction, which we will cover in detail a bit later in this section.



On line 4, we assign the value of `x` to be `281`. This updates the value stored in the memory of `x` from `280` to `281`.

Lastly, on line 5, we assign `y` to the value of `x`. Here, the value stored in the memory object associated with `x` is copied into the memory object associated with `y`.

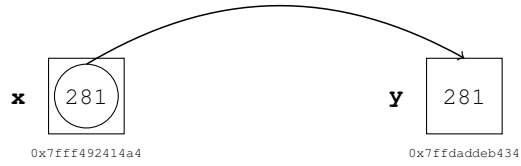


※ 1.2.2 *References and Reference Semantics*

Notice that, when we executed `y = x` in the previous examples, we ended up modifying the *value* of the object that `y` referred to. This is known as *value semantics*, which is the default behavior of initialization and assignment in C++.

`y = x`

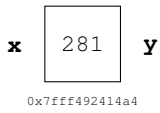
*(**value semantics**: the value of `y` is set to the value of `x`)*



An alternative method of initialization and assignment is *reference semantics*, which changes the object the assigned variable refers to rather than the value of the object itself.

`y = x`

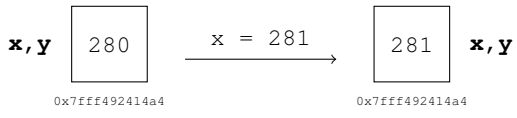
*(**reference semantics**: the object that `y` refers to is set to the object that `x` refers to)*



In C++, reference semantics is only supported during the initialization of a variable. To initialize a variable as a reference, place an ampersand (`&`) to the left of the variable name and assign it to the object you want the variable to refer to. An example is shown in the code below:

```
1  int main() {
2    int x = 280;
3    int& y = x; // y is a reference to x
4    x = 281;
5    std::cout << y << std::endl; // prints 281
6  } // main()
```

Here, a reference is created on line 3 with the declaration `int& y = x`. You can think of this as creating an alternative name `y` for the existing variable `x`. Any changes made to either `x` or `y` will end up changing the exact same object in memory, regardless of which variable name is used to perform the change. For example, when `x` is assigned to a new value of `281` on line 4, the value of `y` also becomes `281`, as `x` and `y` refer to the same integer object in memory.



It is important to note that a reference in C++ cannot be reassigned once it is created! Once you initialize a variable as a reference, that variable cannot be associated with a different memory object until the reference variable goes out of scope. For instance, we would not be able to assign `y` to another object once we initialized it as a reference to `x`, unless that variable `y` goes out of scope (we will cover scope in more detail in a later section of this chapter).

There are many instances where having a reference may be useful. One particularly important use case of references in 281 occurs when working with function parameters. As an example, consider the following function, which attempts to increment an integer counter by one.

```
1  void increment(int counter) {
2    counter += 1;
3  } // increment()
4
5  int main() {
6    int c = 280;
7    increment(c);
8    std::cout << c << std::endl; // prints 280
9  } // main()
```

However, since C++ uses value semantics by default, the variable `counter` within the scope of the `increment()` function is actually initialized as a *copy* of the variable `c` in `main()` on line 7. As a result, a copy of `c` is incremented in the `increment()` function rather than `c` itself! Since the copy goes out of scope within the `increment()` function without being returned, the original `c` does not get modified at all, and line 8 prints out `280`.

To ensure that the counter is actually modified in our function, we would need to pass it in by reference.

```
1  void increment(int& counter) { // pass by reference
2    counter += 1;
3  } // increment()
4
5  int main() {
6    int c = 280;
7    increment(c);
8    std::cout << c << std::endl; // prints 281
9  } // main()
```

Notice the ampersand that we placed on line 1 — this essentially indicates that the `counter` parameter should be a *reference* to whatever object is passed into that function. Therefore, when `c` is passed into the `increment()` function on line 7, `counter` is set as a *reference* to `c`, and any modifications to `counter` are also reflected in `c`. The value of `counter` is incremented to `281`, so the value of `c` also becomes `281`, which gets printed out on line 8.

In general, if you want objects to be modified by a function, you must pass them in by reference. However, passing by reference is not limited to this specific use case. Even if you do not want an object to be modified, it may still be prudent to pass the object in as a *constant* reference (as shown below). This is because passing an object by reference avoids the overhead of copying the object whenever it is passed into the function, which can be inefficient if the object is large.

```
1  void func(const BigObject& obj) { // pass by const reference
2    // do work...
3  } // func()
4
5  int main() {
6    BigObject my_big_object{};
7    func(my_big_object); // my_big_object is not copied into func() because it is passed by reference
8  } // main()
```

Remark: When passing parameters into a function, the following steps provide a good framework for determining whether the parameters should be passed by value or reference.

1. If a parameter should be modified by the function, it should be passed by reference.
2. Else, if a parameter is a primitive type (like an integer or character), pass it by value.
3. Else, if it is an object, pass it by `const` reference.

Note that passing a primitive type (see section 1.3) by reference can be *less* efficient than passing it by value. This is because passing an object by reference is akin to passing a pointer to that object behind the scenes. This not only adds a level of unnecessary indirection for accessing a primitive object (since we have to follow a pointer to where the object is located, instead of accessing it directly), but it also takes up additional memory on the function call stack because the sizes of primitive types (e.g., up to 4 bytes for `int`, 8 bytes for `double`, 1 byte for `char` and `bool`) are typically no larger than the size of pointers (8 bytes on a 64-bit machine).

※ 1.2.3 *Pointers*

When you create an object in C++, it is placed at some memory address during runtime. This specific address, however, is generally not under the programmer's control. If you run the same program more than once, its objects may be placed at different memory locations on each run. However, you can always query the address of an object after it has been created using the "address-of" operator, denoted using an ampersand.

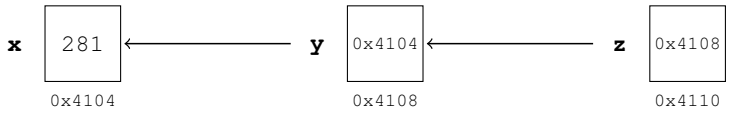
```
1  int main() {
2    int x = 281;
3    std::cout << &x << std::endl; // sample output: 0x7ffebd5f1fdc
4  } // main()
```

The code above prints out the memory address of the integer `x`. However, addresses can also be stored in a category of objects known as **pointers**. To declare a pointer, simply place an `*` symbol to the left of the variable name during declaration. An example is shown below:

```
1  int main() {
2    int x = 281;
3    int* ptr = &x; // ptr stores the address of x
4    std::cout << ptr << std::endl; // sample output: 0x7ffeccc22c4c
5  } // main()
```

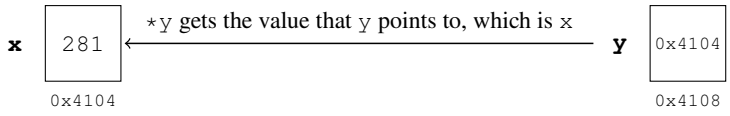
Each data type in C++ has a corresponding pointer type. For instance, an `int*` would represent a pointer to an `int`. Note that pointers are objects as well, and they themselves also have an address in memory. Therefore, it is perfectly valid to have a pointer with the type `int**`, which would represent a pointer to an `int*` (or, in other words, a pointer to pointer to `int`).

```
1  int main() {
2    int x = 281;
3    int* y = &x; // y stores the address of x
4    int** z = &y; // z stores the address of y
5  } // main()
```



You can access the object that a pointer points to by *dereferencing* the pointer using the dereference operator, also denoted with an `*`. In the following code, `y` is a pointer that stores the address of `x`, so applying the dereference operator to `y` (i.e., `*y`) would return the value of `x`.

```
1  int main() {
2    int x = 281;
3    int* y = &x; // y stores the address of x
4    std::cout << *y << std::endl; // prints 281
5  } // main()
```



A few words of warning regarding pointers:

1. If a pointer is not explicitly initialized, it is default initialized with an undefined value. Dereferencing a default-initialized pointer results in undefined behavior!

   ```
   1  int main() {
   2    int *x;
   3    *x = 281; // undefined behavior
   4  } // main()
   ```

2. It is possible for a pointer to outlive the object it is pointing to, which could also result in undefined behavior. For example, the `get_address()` function below returns a pointer to a *local* copy of `x` (because `x` is not passed by reference). This local copy goes out of scope after the function returns, so attempting to dereference its address results in undefined behavior.

   ```
   1  int* get_address(int x) {
   2    return &x; // returns pointer to x, but x goes out of scope after function returns
   3  } // get_address()
   4
   5  int main() {
   6    int val = 280;
   7    int* ptr = get_address(x);
   8    std::cout << *ptr << std::endl; // undefined behavior
   9  } // main()
   ```

In C++, there is a special object known as a *null pointer* that can be used to designate pointers that do not point to a valid object. Any pointer (regardless of type) can be assigned to a null pointer value using the `nullptr` keyword, as shown in the code below.

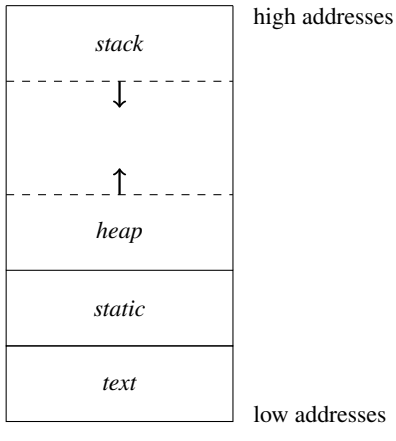
```
1  int main() {
2    int* x = nullptr;
3    double* y = nullptr;
4    char* z = nullptr;
5  } // main()
```

Similar to pointers that are default initialized without a value, dereferencing a `nullptr` also results in undefined behavior.

1.3 Address Space and Dynamic Memory

When you create a new object in a program, memory is allocated to store that object. The general location of this allocation depends on the type of object that was created. Objects whose lifetimes are tied to a particular scope (i.e., local variables) are allocated in a different region of memory than objects that are created dynamically by the programmer.

Every running program (known as a *process*) has its own memory layout, independent from other processes (this independence is maintained by the operating system using something known as *virtual memory*, which is beyond the scope of the class). This memory layout designates the **address space** of a process, which is the range of valid addresses in memory that the process can access. A typical layout for an address space is shown below.



Local variables are stored in the stack region of the address space. The stack grows downward with each allocation (i.e., the first allocation is given the highest address, and subsequent allocations are assigned to lower addresses). Every time a new local variable is declared, space is allocated on the stack to hold that variable. Whenever a local variable goes out of scope, its corresponding memory on the stack is deallocated. *Note that allocation and deallocation for stack memory are both done automatically!*

On the contrary, the heap region of the address space is used to store objects that are allocated dynamically. **Dynamic memory** is allocated *explicitly* by the programmer and is not confined to a particular scope. This type of memory must also be deallocated explicitly as well — failure to do so would cause the memory to remain in the system indefinitely (until it is forceably wiped, such as from a computer shutdown). This is known as a *memory leak*, which can hinder the performance of a system by exhausting the amount of memory available for use.

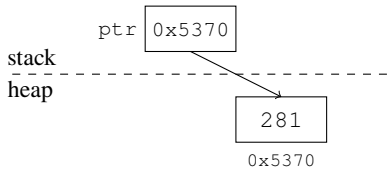
Dynamic memory is allocated with the `new` keyword and deallocated with the `delete` keyword. As such, the lifetime of a dynamically allocated variable starts when it is created with `new` and ends when it is deallocated with `delete`. An example is shown in the code below:

```
1  int main() {
2    int* ptr = new int{281};
3    std::cout << *ptr << std::endl; // prints 281
4    delete ptr;
5  } // main()
```

The `new` keyword does the following:

1. Allocates space for an object of the given type on the heap.
2. Initializes the object with the value of the given initialization expression.
3. Evaluates to the address of the newly created object.

In the code above, line 2 creates a dynamically allocated integer on the heap, initialized with the value 281. Since `new` is used, the expression to the right of the equals sign evaluates to the memory address of this new integer, which is stored as `ptr`. Note that `ptr` is a *local* variable that is allocated on the program stack, and it points to a memory address on the heap! In memory, this would look something like this:



Line 3 dereferences and prints out the value of `ptr`. The dereference (`*`) accesses the value that `ptr` points to (i.e., the value at the address `0x5370` in the illustration above). In this case, this is the integer `281`, so this value gets printed out.

Line 4 applies the `delete` keyword on `ptr`. It is important to note that `delete` deletes the object that `ptr` points to, and not `ptr` itself! In our example above, calling `delete` on `ptr` would deallocate the object at address `0x5370`, or the integer `281`. The `ptr` object still remains on the stack after the delete, since it is a local variable. It should also be noted that `delete` should only be applied to objects that are dynamically allocated using `new`. Attempting to `delete` an object that was not created using `new` results in undefined behavior.

Remark: One notable caveat to this process is the declaration of dynamically-allocated arrays. An array is a fixed sized collection of objects that share the same type, stored contiguously in memory. If you want to allocate an array on the heap, you will need to use `new[]` and `delete[]` with the additional square brackets. An example is shown below:

```
1  int main() {
2    int* arr = new int[5]; // creates array of integers of size 5
3    for (int i = 0; i < 5; ++i) {
4      arr[i] = i; // fills up array with the values [0, 1, 2, 3, 4]
5    } // for
6    delete[] arr; // delete array (notice the extra [])
7  } // main()
```

1.4 Compound Objects

※ 1.4.1 *Primitive Data Types*

When you were first exposed to C++, you were probably introduced to the **primitive data types**: the building blocks of the language. One common primitive data type is the integer (`int`), which is typically 4 bytes and can store integer values between -2,147,483,648 (equal to -2^{31}) and 2,147,483,647 (equal to $2^{31}-1$).[1] Another primitive data type is the character (`char`), which is a 1 byte data type that stores a number, where each number in the range from 0 to 127 represents an ASCII character (for example, the letter 'a' is represented using a `char` value of 97 — this will be covered in more detail in chapter 16).[2] Lastly, a `double` is a data type that can be used to store decimal values; this data type typically requires 8 bytes of memory space. A table of common primitive types is shown below:

Type	Bytes	Minimum Value	Maximum Value
`double`	8	2.22507×10^{-308}	1.79769×10^{308}
`int`	4	-2,147,483,648 (or -2^{31})	2,147,483,647 (or $2^{31}-1$)
`char`	1	0	127
`bool`	1	N/A	N/A

[1] The actual size of an `int` can actually differ in size from system to system, but for our case we will treat it as 32 bits, or 4 bytes. If you want to ensure that an integer object is *exactly* 4 bytes, you can use `int32_t` instead, which we will discuss momentarily.

[2] Although `char` can support values outside the range from 0 to 127, these values do not represent meaningful characters.

Some of these built-in data types can be further modified using a data type modifer that can be used to adjust the size or range of data the type can hold. For instance, the `unsigned` keyword can be used to ensure that a data type is always non-negative (e.g., an `unsigned int` is non-negative and can hold values from 0 to 4,294,967,295, or $2^{32} - 1$). The `short` and `long` keywords can be used to restrict or expand the range that a data type can hold; for instance, a `short int` can only hold values between -32,768 and 32,767 (i.e., -2^{15} to $2^{15} - 1$), while a `long long int` can hold values from -2^{63} to $2^{63} - 1$. The tradeoff of being able to hold such a large number is that more memory is required to store a `long long`, compared to a `short` or normal `int`.

Type	Bytes	Minimum Value	Maximum Value
`unsigned long`	8	0	18,446,744,073,709,551,615 (or $2^{64} - 1$)
`long`	8	-9,223,372,036,854,775,808 (or -2^{63})	9,223,372,036,854,775,807 (or $2^{63} - 1$)
`unsigned int`	4	0	4,294,967,295 (or $2^{32} - 1$)
`int`	4	-2,147,483,648 (or -2^{31})	2,147,483,647 (or $2^{31} - 1$)
`unsigned short`	2	0	65,535 (or $2^{16} - 1$)
`short`	2	-32,768 (or -2^{15})	32,767 (or $2^{15} - 1$)

C++11 introduced the concept of *fixed-width integer types*, defined in `<cstdint>`. These types can be used to guarantee the size of an integer object. For instance, an `int32_t` is an integer that takes up exactly 32 bits (4 bytes). Fixed-width integers can also be unsigned; unsigned versions of these types have the letter `u` prepended to the name of its corresponding signed integer type. For instance, `uint32_t` is an *unsigned* integer that takes up 32 bits. A few fixed-width integer types are shown below.

Type	Bytes	Minimum Value	Maximum Value
`uint64_t`	8	0	18,446,744,073,709,551,615 (or $2^{64} - 1$)
`int64_t`	8	-9,223,372,036,854,775,808 (or -2^{63})	9,223,372,036,854,775,807 (or $2^{63} - 1$)
`uint32_t`	4	0	4,294,967,295 (or $2^{32} - 1$)
`int32_t`	4	-2,147,483,648 (or -2^{31})	2,147,483,647 (or $2^{31} - 1$)
`uint16_t`	2	0	65,535 (or $2^{16} - 1$)
`int16_t`	2	-32,768 (or -2^{15})	32,767 (or $2^{15} - 1$)
`uint8_t`	1	0	255 (or $2^{8} - 1$)
`int8_t`	1	-128 (or -2^{7})	127 (or $2^{7} - 1$)

Another type you may encounter is `size_t`, which is an unsigned integer that is used to represent object sizes. You may often see `size_t` being used for array indexing, loop counting, or size storing. The actual size of a `size_t` object is architecture-dependent, as it is required to be large enough to express the maximum size of any possible object on a given system. For instance, on a 32-bit system, `size_t` will be at least 32 bits wide; similarly, on a 64-bit system, `size_t` will be at least 64 bits wide. For our case, we will treat `size_t` as a type that takes up 64 bits (8 bytes), but you should be aware that this is not always the case depending on the system you are using.

Type	Bytes	Minimum Value	Maximum Value
`size_t`	8	0	18,446,744,073,709,551,615 (or $2^{64} - 1$)

Lastly, when choosing what type of variable to use, think about the possible range of values that you may need to store. If you choose a type with too small of a range, you might end up with *overflow* or *underflow*, in which exceeding the maximum value (or going below the minimum value in the case of underflow) can cause a variable to wrap around to the minimum value (or maximum value for underflow). For instance, the largest number that can be represented using an `int32_t` is 2,147,483,647. Thus, adding 1 to 2,147,483,647 would cause an `int32_t` to wrap all the way around to -2,147,483,648, which is the smallest value the `int32_t` type can hold. On the other hand, if you choose a type with too large of a range, you end up wasting memory by taking up additional space that you never end up needing.

※ 1.4.2 *Structs and Classes*

The types discussed above are fundamental building blocks that can be used to construct bigger things. In many cases, you will need more than these basic types when writing a program. C++ allows you to accomplish this easily with **user-defined data types**, or custom types that you can define to represent objects in your program. For instance, a programmer can define a `Student` object and create `Students` in their program, much like how they would create an `int` or `char`. This is possible using a concept known as a **compound object**, which include structures and classes. An example of a `struct` is shown below:

```
1  struct Student {
2    std::string name;
3    std::string uniqname;
4    int32_t age;
5    int32_t id;
6  };
```

Here, we have introduced the `Student` type, where each `Student` is defined with four member variables: a `string` that stores the student's name, a `string` that stores the student's uniqname, an `int32_t` that stores the student's age, and an `int32_t` that stores the student's student ID. Notice that this is just the *definition* of the `Student` object; it provides a blueprint that allows us to create `Student` objects in our program. To instantiate an instance of a `Student` in memory, we would need to explicitly create the `Student` objects in our program:

```
1  int main() {
2    Student s; // instantiates an instance of a Student
3    ...
4  } // main()
```

In this case, we created a `Student` object, but we did not specify the contents of its member variables. Hence, the contents of the student's name, uniqname, age, and student ID are *default initialized*. This is not good practice, since you may end up with indeterminate values (in other words, the student's age could be default initialized to a junk value, which is likely not something you want). To initialize the members of the `struct` upon initialization, an initializer list can be used, as shown below (both approaches are valid):

```
int main() {
  Student s1 = {"Bobby Tables", "btables", 21, 12345678};
  Student s2{"Bobby Tables", "btables", 21, 12345678};
  ...
} // main()
```

The ordering of the member variables in the initializer list is the same order as the ordering of members in the `struct` declaration. In this case, `name` comes first, then `uniqname`, then `age`, then `id`.

You can also use *designated initializers* to initialize the contents of the object. An example of designated initialization is shown below, where each initialization is specified using a period followed by a designator that names a non-static data member that should be initialized.

```
int main() {
  Student s{
    .name = "Bobby Tables",
    .uniqname = "btables",
    .age = 21,
    .id = 12345678
  };
  ...
} // main()
```

Similar to initializer lists, the order in which members are designated must follow the ordering of members in the `struct` definition.

Individual members of a `struct` can be accessed with the dot (`.`) operator. For example, the following would assign the ID 87654321 to the student and print out this value. This assignment changes the student's ID from 12345678 to 87654321.

```
int main() {
  Student s{"Bobby Tables", "btables", 21, 12345678};
  s.id = 87654321;                                  // assigns 87654321 as ID of s
  std::cout << s.id << std::endl;                   // prints 87654321
  ...
} // main()
```

If we instead had a pointer to an object, we can use the `->` operator to dereference and access a member of that object. For example, if the student were a pointer, the following would accomplish the same result as above:

```
int main() {
  Student s{"Bobby Tables", "btables", 21, 12345678};
  Student* s_ptr = &s;
  s_ptr->id = 87654321;                 // assigns 87654321 as ID of s (what s_ptr points to)
  std::cout << s_ptr->id << std::endl;  // prints 87654321
  ...
} // main()
```

A `class` is another method for defining a custom type in C++. A common misconception is that a `struct` and a `class` are fundamentally different, where `class` objects support member functions while `struct` objects do not. However, this is not true; a C++ `class` is based on a C `struct`, and a `class` and a `struct` are essentially identical. There is only one big difference between a `class` and a `struct`: all members in a `class` are `private` by default, while all members in a `struct` are `public` by default.

As a result, you are able to use a `struct` and `class` interchangeably by just modifying the access level of members. However, in customary usage, it is common to use a `struct` only when you want all members to be public, and a `class` everywhere else. For example, consider the following custom definition of a `String` class. The contents of the `String` object (such as the characters that make up the `String`) are stored as *member variables*, while functions that modify the `String` are defined as *member functions*.

```
class String {
  public:           // public member functions that the user can call on a string
    clear();        // clears the content of the string
    erase(...);     // erases portion of string
    insert(...);    // inserts another string into the original string
    replace(...);   // replaces a portion of the string with new contents
    ...
  private:          // private member variables that store the data of the string
    char* cstr;     // c-string that stores the contents of the string object
    size_t size;    // stores the size of the string object
    ...
};
```

As shown above, we can use the `public` and `private` keywords to specify different levels of access for the user of the class. Members that are `public` can be accessed outside the class, while members that are `private` cannot. Thus, setting members as `private` can prevent users of the object from accessing these member variables directly. This is useful in cases where the class has an invariant that could be potentially broken by the user. For instance, the `size` variable is `private` in the `String` class above because we wouldn't want to give the user of a `String` the ability to modify the `String`'s size! Otherwise, the `String` could end up in an invalid state. For example, a user should not be able to change the size of the string "apple" to something like 563. The `size` variable must be consistent with the contents of the string itself, so its value should be handled internally by the `String` class.

Classes and structs play a major role in the implementation of abstract data types, or ADTs. Recall that **abstraction** is the process of separating *what something does* (the interface) from *how it works* (the implementation). That is, the user of an object is only provided the essential information needed to use the object, and the lower-level implementation details are hidden.

As an example, suppose you wanted to drive a car. When you get in the car and start driving, you only need to know the basic steps required to operate the car. You should know that pressing the accelerator should increase the speed of the car, and that pressing the brakes should slow the car down. But you shouldn't need to know how everything works behind the scenes in order to drive! You don't need to know the mechanics by which pressing the accelerator increases the car's speed; you just need to know that it does. In other words, the details about the car's implementation have been abstracted away.

An **abstract data type (ADT)** behaves in a similar fashion. With an ADT, a user can store data and perform operations on this data, but doing so *requires no knowledge of the implementation of these operations*. In other words, ADTs provide the user with operations and their expected behaviors on stored data, but they do not define implementation details for these operations. Throughout this course, you will be introduced to several ADTs that can be used to perform different functionalities and solve different types of problems.

※ 1.4.3 *Constructors*

A *constructor* can be used to initialize a class-type object, and it is called when an object is created. The syntax for declaring a constructor is very similar to that of a member function, with two main differences: there is no return type, and the name of the constructor is the same as the name of the class. Let's consider a simplified version of the `Student` object defined previously:

```
struct Student {
  std::string name;
  int32_t age;
};
```

To build a constructor that takes in the intended member variables of the class as arguments, you can use a *member-initializer list*. To do so, simply add a colon after the constructor body and list out the member variables in the order they are defined in the object. For each member variable, the value it should be initialized to should be enclosed using parentheses or curly braces. An example is shown below:

```
struct Student {
  std::string name;
  int32_t age;
  Student(std::string name_in, int32_t age_in)
    : name{name_in}, age{age_in} {}
};
```

The above constructor takes in two values, `name_in` and `age_in`, and initializes `name` to the value of `name_in` and `age` to the value of `age_in`. Thus, the following would instantiate a `Student` object with a `name` of `"Alice"` and an `age` of `21`.

```
Student s{"Alice", 21};
```

It is important to make sure that your variables are initialized before you do anything with them in your program, as failing to do so could cause you to accidentally use undefined values! For atomic types, such as integers and doubles, default initialization doesn't "initialize" the type to some predetermined value; instead, it initializes to whatever junk previously existed in memory (i.e., the value is undefined).

If a class defines no constructors at all, the compiler provides an *implicit default constructor*, which default initializes each member variable. If you declared a non-default constructor and want the object to have a default constructor, you would have to explicitly write one. An example of a default constructor is shown below — unlike the custom constructor defined above, the default constructor takes in no arguments.

```
Student()
  : name{"Potato"}, age{281} {}
```

Here, if no argument is provided in the construction of a `Student`, the `Student` object's `name` is automatically initialized to `"Potato"`, and its `age` is automatically initialized to `281`.

※ 1.4.4 *Member Variable Organization*

Consider the following two student objects, each storing the same member variables, but in a different order:

```
struct StudentA {
  std::string name;
  double eecs_281_score;
  int64_t enrollment_timestamp;
  char letter_grade;
  bool has_passed;
  bool has_applied_for_ia;
};
```

```
struct StudentB {
  bool has_passed;
  double eecs_281_score;
  char letter_grade;
  std::string name;
  bool has_applied_for_ia;
  int64_t enrollment_timestamp;
};
```

You might expect these two objects to take up the same amount of space in memory. However, querying the sizes of these objects returns a peculiar result: `StudentA` takes up 56 bytes, but `StudentB` takes up 72 bytes!

```
int main() {
  std::cout << sizeof(StudentA) << std::endl;  // prints 56
  std::cout << sizeof(StudentB) << std::endl;  // prints 72
} // main()
```

Why does `StudentB` use 16 more bytes than `StudentA` despite storing the exact same information? This is due to something known as *struct alignment*, which is beyond the scope of this class (this is an EECS 370 concept). However, you can ensure that your object minimizes its memory footprint by following this simple rule: *when defining a custom object, you should list member variables in order of increasing or decreasing size.* For example, if you wanted to define an object that contains member variables of type `double` (8 bytes), `int` (4 bytes), and `char` (1 byte), you should list out the `double` variables first, followed by `int`, then by `char` (or vice versa):

```
struct MyCustomType {
  // doubles
  // ints
  // chars
};
```

```
struct MyCustomType {
  // chars
  // ints
  // doubles
};
```

Remark: This advice is a bit of an oversimplication, since memory is not the only thing that you should consider when determining the order in which member variables should be listed. As an example, the ordering of members also impacts the order in which the member constructors are run, so if one member must be initialized before another, then that member should be listed first. Style is also important to consider; unless you are creating many instances of a custom object, the amount of memory you save typically is not worth it if you end up reducing the readability or maintainability of your custom object in the process.

1.5 Operator Overloading

In C++, an `operator` is a symbol that can be used to complete a specific function. Every operator in the language has a function-like name (e.g., the + symbol used to add two objects has the name `operator+`). Although operators are predefined for types such as `int` and `double`, they need to be explicitly defined for custom types. The process of defining what an operator does for a custom type is known as *operator overloading*.

Operator overloading is the process of defining the behavior of an operator symbol. Think of this like implementing a function that is invoked using a symbol (like +, <, >) instead of a standard function name. Some common operators that can be overloaded are:

- `operator+`
- `operator-`
- `operator*`
- `operator/`
- `operator=`
- `operator<`
- `operator>`
- `operator<=`
- `operator>=`
- `operator++`
- `operator--`
- `operator==`
- `operator!=`
- `operator+=`
- `operator-=`
- `operator[]`
- `operator()`
- `operator%`
- `operator^`
- `operator<<`
- `operator>>`

To overload an operator for a custom-defined type, you should define the corresponding operator function (e.g., `operator+` for +, `operator<` for <) using the appropriate arguments that the operator is applied on. Let's look at a few examples using a `Point` object introduced below:

```
struct Point {
  int32_t x;
  int32_t y;
  Point()                              // default constructor
    : x{0}, y{0} {}
  Point(int32_t x_in, int32_t y_in)  // custom constructor
    : x{x_in}, y{y_in} {}
};
```

First, let's overload the + operator, where adding two `Point` objects adds both their x- and y-coordinates, e.g.,

```
Point{3, 5} + Point{2, 6} = Point{5, 11}
```

Here, the + symbol works on two different `Point` objects and returns a `Point` with the sum of the two individual points' x- and y-coordinates. Since each operator has a function-like name (the word `operator` followed by the symbol, e.g., `operator+`), we can overload the + operator on `Point` objects as a function that takes in two `Point` objects and returns another `Point` object that sums up the coordinates:

```
Point operator+(const Point& lhs, const Point& rhs);
```

return type — operator name — objects that the operator acts upon

Now, we just need to implement this function to exhibit the behavior we want when two `Point` objects, `lhs` and `rhs`, are added together (`lhs` stands for "left-hand side" and `rhs` stands for "right-hand side"). We will cover the `const` keyword later in this chapter, but just know it prevents the original `Point` objects from being modified.

```
Point operator+(const Point& lhs, const Point& rhs) {
  // return Point with x-coordinate lhs.x + rhs.x and y-coordinate lhs.y + rhs.y
  return Point{lhs.x + rhs.x, lhs.y + rhs.y};
} // operator+()
```

With this, we are able to use the + operator to add two `Point` objects together:

```
int main() {
  Point a{3, 5};      // initialize Point a to (3, 5)
  Point b{2, 6};      // initialize Point b to (2, 6)
  Point c = a + b;    // c = (3, 5) + (2, 6) = (5, 11)
} // main()
```

However, this is not the only way to overload an operator for a custom-defined type. The usual custom is to overload operators within the type definition itself, if possible (an exception would be if the LHS object is not the same type as the class itself). If you overload the operator as a member function, the first argument (in this case, `lhs`) is implicit, and you are able to use the members of the `Point` type directly as if they were part of `lhs` (e.g., `lhs.x` can just be written as `x`). For instance, the following overload of `operator+`

```
Point operator+(const Point& lhs, const Point& rhs) {
  // return Point with x-coordinate lhs.x + rhs.x and y-coordinate lhs.y + rhs.y
  return Point{lhs.x + rhs.x, lhs.y + rhs.y};
} // operator+()
```

can be implemented as a member function within the `Point` definition itself by making the following changes:

```
struct Point {
  int32_t x;
  int32_t y;
  Point()                               // default constructor
    : x{0}, y{0} {}
  Point(int32_t x_in, int32_t y_in)     // custom constructor
    : x{x_in}, y{y_in} {}

  // overloaded operator as a member function of Point
  // lhs is implicit and does not have to be explicitly included (i.e., lhs == *this)
  Point operator+(const Point& rhs) const {
    return Point{x + rhs.x, y + rhs.y};
  } // operator+()
};
```

Some operators, such as +=, may be used to modify the object it is called on. As a result, these operators require you to return an object by reference. If the operator is defined outside the object (i.e., not a member function), you can simply return `lhs` itself:

```
Point& operator+=(Point& lhs, const Point& rhs) {
  // adds the x- and y-coordinates of rhs to lhs and returns a reference to lhs
  lhs.x += rhs.x;
  lhs.y += rhs.y;
  return lhs;
} // operator+=()
```

However, if you define the overloaded function as a member function within the `Point` class, the value of `lhs` is implicit. As a result, to return the value of what would have been `lhs`, you would have to return the current object that invoked the operator. To do so, your return value should be `*this`, as `this` is an internal pointer to the current instance of the class, and dereferencing `this` would get you the current object itself. An example of overloading `operator+=` within the `Point` class is shown below:

```
struct Point {
  int32_t x;
  int32_t y;
  Point()                               // default constructor
    : x{0}, y{0} {}
  Point(int32_t x_in, int32_t y_in)     // custom constructor
    : x{x_in}, y{y_in} {}

  // overloaded operator as a member function of Point
  // lhs is implicit and does not have to be explicitly included (i.e., lhs == *this)
  Point& operator+=(const Point& rhs) {
    x += rhs.x;
    y += rhs.y;
    return *this;
  } // operator+=()
};
```

One advantage of returning `*this` by reference is that it allows you to chain multiple operations together. For example, given three points `a`, `b`, and `c`, the expression `a += b += c` should be valid (and is set `a` to `a + (b += c)`).

As mentioned earlier, if the left-hand side (`"lhs"`) parameter of an operator is not of the same type as the custom object the operator acts upon, then the operator overload cannot be implemented as a member function of that custom object. One example of this can occur if you try to overload `operator<<`. Overloading `operator<<` can be extremely useful, as it allows you to output your custom objects in any format you want. For example, suppose you wanted to print out your `Point` objects in the form `"(x, y)"` for member variables `x` and `y`. The complicated way to do this would be as follows (assuming that `x` and `y` are publicly accessible and `pt` is the `Point` object you want to print):

```
std::cout << "(" << pt.x << ", " << pt.y << ")" << std::endl;
```

If you overload `operator<<`, you can define this behavior for the `Point` object itself. This will allow you to get the same result by just outputting the `Point` object to a stream:

```
std::cout << pt << std::endl;
```

To overload `operator<<`, you need to pass in a reference to `ostream` and return the same `ostream` by reference. The overloaded function also takes in the object type whose output format you are trying to define. An example is shown below:

```
std::ostream& operator<<(std::ostream& os, const Point& pt) {
  // define the output format you want to print for this custom object
  os << "(" << pt.x << ", " << pt.y << ")";
  return os;
} // operator<<()
```

After doing this, the following code would print `"(5, 11)"` since `operator<<` is now defined for `Point` objects (assuming that the overloaded `operator+` from before is still defined):

```
int main() {
  Point a{3, 5};                    // initialize Point a to (3, 5)
  Point b{2, 6};                    // initialize Point b to (2, 6)
  std::cout << a + b << std::endl;  // prints "(5, 11)"
} // main()
```

Similar to returning `*this` when overloading the += operator, it is important to return a reference to the `ostream` when overloading `operator<<`, since this allows multiple calls of `operator<<` to be chained on a single line. Without returning a reference, `cout << a + b` would work, but `cout << a + b << endl` would not, since `operator<<` is used multiple times in the same expression.

1.6 Function Objects and Comparators

Throughout this course, you may need to sort custom objects in your projects based on some defined ordering. For example, you may be tasked with sorting a collection of `Student` objects by age. However, since a `Student` is a custom-defined type, we need to let the program know that `Student` objects should be ordered based on their `age` member variable. One way to accomplish this is to overload the less than (<) operator so that it compares two `Student` objects based on age. An example is shown below:

```
struct Student {
  std::string name;
  int32_t age;
  int32_t id;
  bool operator<(const Student& rhs) const {
    return age < rhs.age;
  } // operator<()
};
```

However, what if you wanted to sort students by name for one step of an algorithm, and by age for another step? The < operator can only be overloaded once per object, so it would not make sense to permanently define name or age as the only method of ordering. To address this, you can use a comparator. A comparator is a function object (functor), which overloads the function call `operator()`. By overloading the function call operator, the object can be called as if it were an ordinary function.

To create a comparator, create a custom object and overload its `operator()` so that it takes in two objects and returns whether one (`lhs`) has a lesser value than the other (`rhs`).

```
struct Student {
  std::string name;
  int32_t age;
  int32_t id;
};

// comparator for Student object
struct StudentComparator {
  // overload operator() so that this object can be used like a function
  bool operator() (const Student& lhs, const Student& rhs) const {
    return lhs.age < rhs.age;
  } // operator()()
};
```

With this definition, we can instantiate a `StudentComparator` object and use it as a function to compare two `Student` objects. An example is shown below:

```
int main() {
  Student s1{"Alice", 21, 12345678};
  Student s2{"Bob", 22, 87654321};
  StudentComparator comp;                     // create function object
  std::cout << comp(s1, s2) << std::endl;     // prints whether s1 < s2
} // main()
```

Since we defined our comparator to compare `Student` objects using `age`, `comp(s1, s2)` evaluates to `true` since 21 < 22.

However, defining the comparator based on age alone may not be enough. What if we have two students with the same age? How would we order these students? If we order students with the same age by name, how would we deal with students with the same name *and* age? Luckily, we can build a comparator that takes in many levels of comparisons to determine the correct ordering of students. To illustrate this, suppose we wanted to build a comparator that enforces the following ordering:

- Students are ordered by age first, with younger students coming before older students.
- If two students have the same sage, ties are broken by name (ordered alphabetically).
- If two students have the same name *and* age, ties are broken by student ID (which is guaranteed to be unique).

Let's look at our original comparator that only considers age:

```
struct Student {
  std::string name;
  int32_t age;
  int32_t id;
};

struct StudentComparator {
  bool operator() (const Student& lhs, const Student& rhs) const {
    return lhs.age < rhs.age;
  } // operator()()
};
```

We want to modify this comparator so that, if two students have the same age, the comparator compares their names instead. This can be done by adding an `if` check so that `name` is considered if the values of the two `Student` objects' ages are the same.

```
struct StudentComparator {
  bool operator() (const Student& lhs, const Student& rhs) const {
    if (lhs.age == rhs.age) {
      return lhs.name < rhs.name; // if ages are equal, use name to determine order
    } // if
    return lhs.age < rhs.age;
  } // operator()()
};
```

We can add one final check if both name and age are the same. In this case, the comparator should compare the IDs of the students:

```
struct StudentComparator {
  bool operator() (const Student& lhs, const Student& rhs) const {
    if (lhs.age == rhs.age) {
      if (lhs.name == rhs.name) {
        return lhs.id < rhs.id; // if names are also equal, ID determines order
      } // if
      return lhs.name < rhs.name;
    } // if
    return lhs.age < rhs.age;
  } // if
};
```

We have successfully built a comparator that can order two `Student` objects, based on `age` first, then `name`, then `id`. If we sorted a collection of `Student` objects using this comparator, all the students would be first sorted in increasing order of age. Students with the same age would then be sorted in alphabetical order, and students with the same name and age would be sorted in increasing order of student ID.

```
Student s1{"Alice", 20, 12345678};
Student s2{"Alice", 21, 23456789};
Student s3{"Alice", 21, 34567890};
StudentComparator comp;
std::cout << comp(s1, s2) << std::endl;   // evaluates to true, since s1 < s3
std::cout << comp(s2, s3) << std::endl;   // evaluates to true, since s2 < s3
```

Function objects are also useful in ways beyond sorting a container of custom objects. Comparators and other functors can be integrated with many common standard library functions (which we will cover in greater depth in chapter 11). For instance, suppose you are given a container of objects, and you wanted to remove all objects in the container that satisfy a given condition. To accomplish this, you can create a function object that returns `true` if an object satisfies the condition required for removal and `false` otherwise. Then, you can pass this function object into a C++ standard library algorithm that can help remove all objects within a range that satisfy the condition.

Because function objects can be implemented as a `struct` or `class`, they can be used to hold state through the use of member variables. For instance, consider the following function object:

```
1  struct Student {
2    std::string name;
3    double exam_score;
4  };
5
6  struct ExamPass {
7  private:
8    double threshold;
9
10 public:
11   ExamPass(double thres_in) : threshold{thres_in} {}
12
13   bool operator() (const Student& s) const {
14     return s.exam_score > threshold;
15   } // operator()()
16 };
```

This function object returns `true` if a student's exam score is higher than some given threshold and `false` otherwise. In this example, we are able to set the threshold to any value we want when instantiating the function object:

```
1  int main() {
2    Student s{"Alice", 45};
3    ExamPass pass{50};                    // threshold set to 50
4    std::cout << pass(s) << std::endl;    // prints 0 for false
5  } // main()
```

```
1  int main() {
2    Student s{"Alice", 45};
3    ExamPass pass{40};                    // threshold set to 40
4    std::cout << pass(s) << std::endl;    // prints 1 for true
5  } // main()
```

1.7 Scope and Namespaces

※ 1.7.1 *Scope*

When you declare a variable, function, or type in C++, where can you use it? The answer to this is determined by the **scope** of the named object in question. The scope of an object is the region of the program in which a named object can be used. An object's scope is defined at compile time, so attempting to use an object that is out of scope will result in a compile error.

Most of the variables you will encounter in this class will have local, or block scope. The scope of a variable with local scope (which we call a *local variable*) begins when the variable is created, and it ends at the closing brace that concludes the block of code that the variable was defined in. For instance, the variable `val` can only be used after the point of declaration, but within the curly braces that it is declared in (i.e., after line 4, but before the closing brace on line 6):

```
1  int main() {
2    // cannot use val here (not declared)
3    {
4      int32_t val = 281;
5      // can use val here (within scope)
6    }
7    // cannot use val here (out of scope)
8  } // main()
```

In a single scope, each variable name must uniquely represent a single entity. That is, there cannot be more than one object within the same scope that share the same name:

```
1  int main() {
2    int32_t val = 281;
3    int32_t val = 370; // not allowed!
4  } // main()
```

However, if there are nested inner scopes within an outer scope, an inner scope can reuse a name that exists in its outer scope to reference a completely different entity. When this happens, the inner scope will only have access to the variable that it declared within its own scope, and not the variable with the same name in an outer scope:

```
1  int main() {
2    int32_t val = 281;
3    {
4      // inner scope
5      int32_t val = 370;
6      std::cout << val << std::endl;  // prints 370
7    }
8    std::cout << val << std::endl;    // prints 281
9  } // main()
```

Variables that are initialized within the introduction of a statement (such as the variables created in a loop condition) are local to the block they introduce. For example, the value of `i` within the `for` loop is equal to the running counter and not the value `281`. This is because the `i` declared in the introduction of the `for` loop is local to the scope of the `for` loop, which is the region between the curly braces on lines 3 and 5. Thus, the following code prints out `"0 1 2 3 4"` instead of `"281 281 281 281 281"`.

```
1  int main() {
2    int32_t i = 281;
3    for (int32_t i = 0; i < 5; ++i) {
4      std::cout << i << " ";
5    } // for i
6  } // main()
```

Scope also applies to items within a `class` or `struct`. The scope of any member of a class extends throughout the entire class, regardless of where the member is declared. Similar to block scope, each name within a class can only be used to represent a single member, regardless of whether that member is a variable or a function.

```
1  class MyClass {
2    int32_t sum;
3
4    // not allowed, since sum is the name of another member of the same class
5    int32_t sum(int32_t a, int32_t b) {
6      return a + b;
7    } // sum()
8  };
```

If an object is declared outside any class or function, the object has global scope. The scope of a variable with global scope (which we call a *global variable*) begins when the variable is created and extends all the way to the end of the file in which it is declared. The scope of a global variable can also be extended to other files in a project using the `extern` keyword (which you won't need to know about for this class).

```
1  int32_t VAL = 281;     // global variable
2
3  void func() {
4    std::cout << VAL << std::endl;
5  } // func()
6
7  int main() {
8    VAL = VAL + 1;       // VAL is now 282
9    func();              // prints 282
10 } // main()
```

※ 1.7.2 *Namespaces* (✻)

Because entities in the same scope cannot share the same name, you could have a problem if you have a large project that declares many different functions, variables, and types. This can be a nuisance if you are working with a codebase that depends on a lot of external libraries, since there is a chance you will end up with multiple variables across different libraries that share the same name. For example, suppose you have a project that utilizes two different libraries, library A and library B. Library A has a global function called `do_something()`, which you use in your program. Now, suppose you upgrade to a new version of library B, which introduced its own `do_something()` function in its global namespace. As a result, you have a naming conflict, and your entire project would no longer compile!

To resolve this issue, we can use *namespaces* to group entities that otherwise would have been part of the same global scope into smaller scopes. This smaller scope is known as a *namespace scope*. An example is shown below:

```
1  namespace foo
2  {
3    void do_something() { /*...*/ }
4  } // namespace foo
5
6  namespace bar
7  {
8    void do_something() { /*...*/ }
9  } // namespace bar
10
11 int main() {
12   foo::do_something();  // run version of do_something() in foo namespace
13   bar::do_something();  // run version of do_something() in bar namespace
14 } // main()
```

Namespaces are useful for preventing name collisions, as shown above. To specify a particular namespace scope, you can use the *scope resolution operator* (::). Alternatively, you can avoid this by using the `using` keyword:

```
1  int main() {
2    {
3      using namespace foo;
4      do_something();       // runs foo::do_something()
5    }
6    {
7      using namespace bar;
8      do_something();       // runs bar::do_something()
9    }
10 } // main()
```

You can also specify which names you want to take from each namespace:

```
int main() {
  {
    using foo::do_something;
    do_something();            // runs foo::do_something()
  }
  {
    using bar::do_something;
    do_something();            // runs bar::do_something()
  }
} // main()
```

You may have recognized namespaces from the term `using namespace std;` which is included at the top of many source files you have seen so far. This is because all files in the C++ standard library implement their functionality within the `std` namespace! This is why you need to prepend `std::` in front of standard library entities such as `std::string` and `std::vector<>` if you do not specify the `std` namespace otherwise. There are other namespaces that are available; for example, the Boost library can be found in the `boost` namespace.

Remark: You should avoid including the line `using namespace std;` in a header file! Even though it may save you some typing time, blanket including a namespace can be dangerous — if anyone else includes your header file in their code, you would essentially force them to use all the namespaces you included, which could render their code unusable. This rule also applies for other namespaces, not just `std`.

1.8 Common C++ Keywords

※ 1.8.1 *The `const` Keyword*

In the previous few sections, the keyword `const` was used to qualify certain variables and functions. A variable that is qualified using the `const` keyword cannot be modified.

```
const int32_t MAX_VALUE = 281;
MAX_VALUE = 5; // not allowed!
```

The `const` keyword can also be used with pointers. Consider the following pointer declaration:

```
int32_t* num = new int32_t{281};
```

There are two things we can do to modify this pointer. First, we could modify the content that the pointer points to (e.g., change `num` so that it points to a value of `280` instead of `281`):

```
*num = 280;
```

We can also change the memory address that the pointer is pointing to:

```
int32_t* num2 = new int32_t{280};
num = num2;
```

If the `const` keyword is placed before the pointer symbol, you prevent the *content* that the pointer points to from being modified. However, changing the pointer itself is allowed.

```
const int32_t* num = new int32_t{281};
*num = 280; // not allowed!
```

You can also add the `const` keyword after the pointer symbol. This would allow you to change the content that the pointer points to, but it disallows you from reassigning the pointer to another address.

```
int32_t* const num = new int32_t{281};
num = nullptr; // not allowed!
```

To prevent modification of *both* the content of the pointer and the pointer itself, you would need to add `const` in both places:

```
const int32_t* const num = new int32_t{281};
*num = 280;    // not allowed!
num = nullptr; // not allowed!
```

Remark: One of the biggest stylistic debates in C++ involves the placement of the `const` modifier relative to the type that needs to be kept constant. There are two primary camps: west and east const. West const (or const west) is the format used in the notes so far, where the `const` qualifier is placed to the left of what it modifies:

```
// const int (int cannot be modified)
const int32_t val = 281;
// const int reference (int cannot be modified)
const int32_t& ref = val;
// pointer to const int (int cannot be modified, but pointer can)
const int32_t* ptr1 = &val;
// const pointer to int (pointer cannot be modified, but int can)
int32_t* const ptr2 = &val;
// const pointer to const int (neither point nor int can be modified)
const int32_t* const ptr3 = &val;
```

West const is more widespread, but it is less consistent and requires you to remember rules regarding which object is const and which isn't. This is where east const comes into play. With east const, the `const` keyword is placed to the right of what it modifies:

```
// const int (int cannot be modified)
int32_t const val = 281;
// const int reference (int cannot be modified)
int32_t const& ref = val;
// pointer to const int (int cannot be modified, but pointer can)
int32_t const* ptr1 = &val;
// const pointer to int (pointer cannot be modified, but int can)
int32_t* const ptr2 = &val;
// const pointer to const int (neither point nor int can be modified)
int32_t const* const ptr3 = &val;
```

Here, there is one consistent rule: the type that cannot be modified can always be determined by looking at what is left of the `const` keyword. For instance, `int32_t const*` indicates that the `int32_t` cannot be modified, since `int32_t` is what is left of `const`. Similarly, `int32_t* const` indicates that the `int32_t*` (i.e., the pointer) cannot be modified, since it is left of `const`.

Whether you use west or east const is a matter of preference, and there are arguments for both sides. The recommended suggestion, however, is to be consistent with the style you use. If you are writing a program, try to avoid mixing and matching the two styles in your code, since that typically makes your code less readable. The same applies when working with existing code in a large codebase; if you are modifying an existing file, try to adhere to the same stylistic choices that are used (any style changes should be reflected in the entire file).

For these notes, west const will be used. This is not because west const is inherently superior; rather, it is because this style is more prevalent in existing codebases, so it is important for you to be familiar with how it works. However, feel free to use east const in your own code if you feel that it is the better style.

Let's go back and consider the following `Student` object that we defined earlier:

```
1  struct Student {
2    std::string name;
3    int32_t age;
4    int32_t id;
5    bool operator<(const Student& rhs) const {
6      return age < rhs.age;
7    } // operator<()
8  };
```

Notice how our overloaded `operator<` operator has two uses of `const`: one in the type definition of `rhs`, and one right before the starting curly brace of the function definition. What is the purpose of each of these uses of `const`?

Here, we pass the `Student` object `rhs` by reference. This allows us to access the `Student` object in our overloaded `operator<` function without making a separate copy of a `Student` object (which would have happened had the `Student` been passed by value). If you are trying to pass a large object into a function, you generally want to pass it in by reference. This is because passing by value would require you to make a copy of the object every time the function is called, which is inefficient in terms of both time and memory. By specifying that `rhs` is a `const Student`, we indicate that the function `rhs` is passed into (in this case, `operator<`) *cannot* modify the contents of `rhs`.

```
1  struct Student {
2    std::string name;
3    int32_t age;
4    int32_t id;
5    bool operator<(const Student& rhs) const {
6      rhs.age = 20; // not allowed!
7      return age < rhs.age;
8    } // operator<()
9  };
```

What about the second `const` keyword that occurs after all the parameters? If this `const` keyword is specified for a member function of an object, it indicates that *the member function is not allowed to modify any member variables of the object itself.* In the case of the overloaded `operator<` function, the second `const` indicates that the `operator<` function cannot modify any member variables of the `Student` object that invokes it (i.e., `name`, `age`, and `id`).

```
1  struct Student {
2    std::string name;
3    int32_t age;
4    int32_t id;
5    bool operator<(const Student& rhs) const {
6      age = 20; // not allowed!
7      return age < rhs.age;
8    } // operator<()
9  };
```

If you have a member function of an object that does not modify the member variables of the object itself, it is important to specify the member function as `const`. This is important, not only because it prevents accidental changes to the object, but also because **non-`const` member functions can only be called by non-`const` objects**! On the other hand, `const` member functions can be called by any type of object, regardless of whether the object itself is `const`. Consider the following code:

```
1  struct Student {
2    std::string name;
3    int32_t age;
4    int32_t id;
5    bool operator<(const Student& rhs) {
6      return age < rhs.age;
7    } // operator<()
8  };
9
10 void print_smaller_age(const Student& s1, const Student& s2) {
11   if (s1 < s2) {
12     std::cout << s1.age << std::endl;
13   } // if
14   else {
15     std::cout << s2.age << std::endl;
16   } // else
17 } // print_smaller_age()
18
19 int main() {
20   Student s1{"Alice", 20, 12345678};
21   Student s2{"Bob", 21, 87654321};
22   print_smaller_age(s1, s2);
23 } // main()
```

The `print_smaller_age()` function prints out the smaller age of two students that are passed in. However, this code does not work because `operator<` is not defined as `const`. Since `s1` and `s2` are both `const Student` objects, `operator<` cannot be used to compare the two students since the `operator<` function itself is not declared as `const`. The compiler doesn't know whether `operator<` will modify any member variables since it is defined as non-`const`, so when `const Student s1` attempts to call its overloaded `operator<` function, the compiler prevents it from happening since it cannot guarantee that `s1` will not be modified.

To fix this issue, simply define the overloaded `operator<` function as `const`, so that the function can be called by both `const` and non-`const Student` objects.

```
1  struct Student {
2    std::string name;
3    int32_t age;
4    int32_t id;
5    bool operator<(const Student& rhs) const { // specify function as const
6      return age < rhs.age;
7    } // operator<()
8  };
9
10 int main() {
11   Student s1{"Alice", 20, 12345678};
12   Student s2{"Bob", 21, 87654321};
13   print_smaller_age(s1, s2); // successfully prints 20
14 } // main()
```

※ 1.8.2 *The `constexpr` Keyword* (✻)

Another keyword that you may encounter is `constexpr`, which stands for constant expression. This keyword should be applied to values that are constant and *known during compilation*. When used appropriately, `constexpr` may improve the performance of a program, since computing a value during compile time removes the computation work from runtime. There are also cases where `constexpr` is needed to perform functionality that is not allowed using runtime values, such as declaring the size of an array, as shown below.

```
1  int main() {
2    constexpr size_t array_size = 281;
3    int32_t my_arr[array_size]; // array of size 281
4  } // main()
```

The `constexpr` keyword is particularly interesting when applied to functions. A `constexpr` function returns compile-time constants if the arguments it is called with are also compile-time constants. Otherwise, if any one of its arguments has an unknown value during compilation, it behaves like a normal function and computes its result during runtime. For instance, consider the following `degrees_to_radians()` function, which is qualified as `constexpr`:

```
1  constexpr double pi = 3.14159265358979323846;
2  constexpr double degrees_to_radians(double angle) {
3    return angle * (pi / 180.0);
4  } // degrees_to_radians()
```

If the value of `angle` is known during compile time, the result of the function will also be calculated during compile time (i.e., when the program compiles). If the value of `angle` is not known during compile time, the result of the function will be calculated during runtime.

```
1  int main() {
2    constexpr double radian1 = degrees_to_radians(57.296); // computed while compiling
3    double degree_from_file = read_value_from_file();
4    double radian2 = degrees_to_radians(degree_from_file); // computed during runtime
5  } // main()
```

User-defined types can also be determined during compilation if their constructors and member functions are qualified as `constexpr`. An example of such a class declaration is shown below:

```
1  class Foo {
2    int32_t i;
3    double j;
4  public:
5    constexpr Foo(int32_t i_in, double j_in)
6      : i{i_in}, j{j_in} {}
7
8    constexpr int32_t get_i() const {
9      return i;
10   } // get_i()
11
12   constexpr double get_j() const {
13     return j;
14   } // get_j()
15 };
16
17 constexpr Foo add_foos(const Foo& f1, const Foo& f2) {
18   return Foo{f1.get_i() + f2.get_i(),
19              f1.get_j() + f2.get_j()};
20 } // add_foos()
21
22 int main() {
23   // all done during compile time
24   constexpr Foo f1{183, 2.8};
25   constexpr Foo f2{281, 3.7};
26   constexpr Foo f3 = add_foos(f1, f2); // f3 = Foo{464, 6.5};
27 } // main()
```

Here, the `Foo` constructor is marked as `constexpr`. Thus, if the arguments of the constructor are known during compilation, a `Foo` object can be constructed at compile time if qualified as `constexpr` (as shown on lines 23 and 24). Similarly, because `Foo::get_i()` and `Foo::get_j()` are `constexpr` functions, their return values are computed during compilation if they are invoked on a `constexpr Foo` object. This allows us to write functions that can return `constexpr Foo` objects using these getter functions, as shown by the `add_foos()` function on line 17. Putting this all together, the above code in `main()` to be computed entirely while compiling the program (rather than during program runtime)! In general, migrating computations from runtime to compile time will make a program run faster (and is often a preferable decision to make, even if it comes at the cost of longer compilation times).

※ 1.8.3 *The `static` Keyword*

The `static` keyword has three main use cases: it can be used to define

1. a variable inside a function
2. a variable inside a `class` or `struct` definition
3. a global variable that is defined in one file of a multifile program

A *static local variable* is a local variable declared with the `static` keyword within a function. If a variable is declared as `static` in a function, there exists only one copy of the variable, and it remains in memory and maintains its value for the entire duration of the program. In the example below, the function calls to `foo()` all use the same instance of `counter`.

```
1  int32_t foo() {
2    static int32_t counter = 0;
3    counter += 1;
4    return counter;
5  } // foo()
6
7  int main() {
8    std::cout << foo() << std::endl; // prints 1
9    std::cout << foo() << std::endl; // prints 2
10   std::cout << foo() << std::endl; // prints 3
11 } // main()
```

If `static` is used to define a member variable of an object, that member variable has the same value for any instance of that object. In other words, without the `static` keyword, each object of the `class` or `struct` would have *its own copy* of the member variable. However, if a member variable is declared as `static`, its value is shared by all instances of the object.

In order to access a `static` member variable outside the class, the scope-resolution operator (`::`) must be used. An example is shown below:

```
struct Foo {
  static int32_t counter;
  void print_count() {
    counter += 1;
    std::cout << counter << std::endl;
  } // print_count()
};

int32_t Foo::counter = 0; // initializes counter for all Foo objects to 0

int main() {
  Foo f1;
  Foo f2;
  f1.print_count();      // prints 1
  f2.print_count();      // prints 2
} // main()
```

Lastly, it is also possible to use the `static` keyword before a global variable. By doing so, the scope of the `static` global variable is limited to the file it is defined in — code in other files cannot access the variable, even if they are part of the same project.

※ 1.8.4 *The `mutable` Keyword*

As mentioned previously, if a member function of an object is declared as `const`, it cannot modify any of the object's own member variables. However, in some cases, we may want a member variable to be modified even if the member function we call promises not to change anything. Consider the following:

```
class ProjectScoreTracker {
  double project_average;       // current project average
  ...                           // other data
  int32_t request_counter;      // counts number of times scores are requested
public:
  ProjectScoreTracker() : request_counter{0} {}
  void request_scores() const {
    // prints out project score data
    ...
    // update request counter
    ++request_counter;
  } // request_scores()
};
```

The class `ProjectScoreTracker` keeps track of data for an EECS project, and the `request_counter` member variable keeps track of the number of times that project scores are fetched. However, this code does not compile, since `request_scores()` is attempting to modify a class member variable even though the function is declared as `const`. If we try to remove the `const` declaration from the function, then it cannot be called if the `ProjectScoreTracker` we want to call the function on is `const`.

One way to fix this issue is to use the `mutable` keyword. If you declare a member variable of an object as `mutable`, you give `const` member functions of that object permission to modify that member variable. The following code would compile:

```
class ProjectScoreTracker {
  double project_average;           // current project average
  ...                               // other data
  mutable int32_t request_counter;  // counts number of times scores are requested
public:
  ProjectScoreTracker() : request_counter{0} {}
  void request_scores() const {
    // prints out project score data
    ...
    // update request counter
    ++request_counter;
  } // request_scores()
};
```

Here, by declaring the `request_counter` member variable as `mutable`, we give the `request_scores()` function permission to modify it. The function, however, cannot modify any of the other member variables (such as `project_average`) since those variables are not declared as `mutable`.

Obviously, this example is a bit contrived, and there are ways to get around this error without relying on the `mutable` keyword. However, depending the projects that are assigned during your semester, there may be instances where knowledge of this keyword will be useful. One such scenario occurs if you have a priority queue of custom objects, and you want to modify a member variable of the object at the top of the priority queue in a way that does not disrupt its priority (we will cover priority queues in chapter 10, so there is no need to worry about this information yet if you do not know what it is). However, the STL's priority queue returns a `const` reference to its top element, and you cannot modify the element at the top without having to pop it first, modify it, and then push it back into the queue. By defining the member variable you want to modify as `mutable`, you can avoid having to pop and repush in the object you want to modify and instead update it in-place (but this is only safe as long as your modification does not disrupt the existing priority of objects in your priority queue).

The `mutable` keyword can used for variables that do not keep track of the actual data of an object, but rather other useful information about the data that can be modified. Because of this, if both the information about the data and the data itself are encapsulated within the same class, `mutable` allows the former to be modified without breaking the invariant that the latter cannot.

※ 1.8.5 *The `explicit` Keyword*

If an object has a constructor that takes in a single argument with a type *different* from the type of the object itself, then the constructor is known as a *conversion constructor*. For example, consider the following:

```
1  class Foo {
2    int32_t foo_x;
3  public:
4    Foo(int32_t input) : foo_x{input} {}
5  };
```

Here, the constructor on line 4 takes in an integer and assigns its value to the member variable `foo_x`. As a result, if you attempt to use the conversion constructor to initialize a `Foo` object from an integer, the constructor simply sets the value of `foo_x` to that integer:

```
Foo myFoo{3}; // sets the value of foo_x to 3
```

A conversion constructor is perfectly fine if the programmer wants to convert objects of one type to objects of another type. However, single-argument conversion constructors can be inappropriately used by the compiler to make *implicit type conversions behind the scenes*, or type conversions that are made by the compiler without prompt from the user. The compiler is not the programmer, so it shouldn't make decisions on the programmer's behalf! There are cases where an implicit conversion goes against the intents of the programmer, and unwanted implicit conversions are often the toughest bugs to track down.

The following is an example of an implicit conversion. The programmer doesn't directly call the constructor to make the conversion, but the compiler does the conversion on the programmer's behalf because it thinks the programmer intended to convert an integer to a `Foo` object:

```
Foo myFoo = 3; // compiler uses ctor to convert 3 to object of type Foo
```

The `explicit` keyword can be used to remove control from the compiler and give it back to the programmer. If a single argument constructor is defined as `explicit`, then the programmer must *explicitly* call that constructor if they want to perform a conversion from one type to another. In other words, the compiler cannot use the explicit constructor to make implicit conversions behind the scenes, without the approval of the programmer. Using the above example, the following redefinition of the `Foo` class prevents unintended implicit conversions:

```
1  class Foo {
2    int32_t foo_x;
3  public:
4    explicit Foo(int32_t input) : foo_x{input} {}
5  };
```

After the `explicit` keyword is added, the following will be true:

```
Foo myFoo{3};       // okay since ctor explicitly called, so foo_x is set to 3
Foo myFooToo = 3;   // NOT okay: compiler error since implicit conversion
```

Below is an practical example where failing to include `explicit` can cause a major problem. Consider the following class, which implements a custom string object:

```
1  class CustomString {
2    ...                            // member variables
3  public:
4    CustomString(int32_t n);       // constructs CustomString with size n
5    CustomString(const char* p);   // initialize CustomString with char*
6    ...
7  };
8
9  void print(CustomString c);      // prints the CustomString
```

Suppose there are two ways to construct this `CustomString` object. If an integer is passed into the constructor, a `CustomString` is created with its size set to that integer (using the constructor on line 4). If a `const char*` is passed into the constructor, a `CustomString` is created with the contents of the `const char*` (using the constructor on line 5). The `print()` function prints the contents of the `CustomString` it is given. Now, suppose a programmer calls the following:

```
1  // print "EECS 281"
2  print("EECS");
3  print(281);
```

What happens here? The first `print()` line runs without issue, and `"EECS"` is printed out. However, the second `print()` line contains a bug: the programmer intended to call `print("281")` but forgot the quotation marks! The compiler notices that `print()` requires an argument of type `CustomString`, but it instead received an argument of type `int32_t`. But wait! There is a single-argument constructor defined on line 4 that can construct a `CustomString` using an integer, and this constructor is not `explicit`! As a result, the compiler implicitly converts the `281` into a `CustomString` using the constructor on line 4 and passes the resulting `CustomString` into the `print()` function. This actually ends up constructing an empty `CustomString` with size 281 and not a `CustomString` initialized to `"281"`, so the program ends up printing an empty string of size 281. This was not the intent of the programmer, and the compiler does not issue any errors that may notify the programmer of this bug.

Defining the constructor on line 4 as `explicit` would prevent this from happening. Without this keyword, the compiler would be able to make implicit conversions without any knowledge of the programmer, which can make debugging much more painful!

```
1  class CustomString {
2    ...                                  // member variables
3  public:
4    explicit CustomString(int32_t n);    // constructs CustomString with size n
5    CustomString(const char* p);         // initialize CustomString with char*
6    ...
7  };
8
9  void print(CustomString c);            // prints the CustomString
```

1.9 Templates

※ 1.9.1 *Templates*

C++ is a statically typed language, which means the types of variables must be known at compile time. The need for variable types to be fixed at compile time brings up a notable issue: how can the same code be adapted for different types? Consider the following `swap()` function:

```
1  void swap(int32_t& a, int32_t& b) {
2    int32_t temp = a;
3    a = b;
4    b = temp;
5  } // swap()
```

This code allows you to swap the values of two integers that are passed in. However, what if you wanted to swap two `doubles` instead? This function would fail since a reference to `int32_t` cannot be set to a reference to `double`. As a result, this particular `swap()` function would be pretty useless on anything that is not an `int32_t`!

Instead of rewriting the `swap()` function on all possible types, we can instead use *templates*. Templates allow us to do *generic programming*: writing code that applies to many different types. The compiler uses the templates to modify the code as needed for the type we want. To illustrate how this works, let's go back to the `swap()` example we had before. To allow this function to work on all types, we can define the following function template:

```
1  template <typename T>
2  void swap(T& a, T& b) {
3    T temp = a;
4    a = b;
5    b = temp;
6  } // swap()
```

Here, the letter `T` represents the *type parameter*. The compiler deduces what the type of `T` actually is from the type that is passed into the arguments of the function call, and it replaces the type parameter `T` with what the type actually is. The compiler must see the template definition first, *before* it is used in any code that you write! As an example, suppose you call the swap function with two integers:

```
1  int32_t a = 5;
2  int32_t b = 6;
3  swap(a, b);
```

The compiler sees from the function call that two objects of type `int32_t` were passed into the `swap()` function, so it will use the `swap()` template and replace all instances of `T` with `int32_t`.

```
1  template <typename T>
2  void swap(T& a, T& b) {
3    T temp = a;
4    a = b;
5    b = temp;
6  } // swap()
```

```
1  // Compiler deduces that T is int32_t
2  void swap(int32_t& a, int32_t& b) {
3    int32_t temp = a;
4    a = b;
5    b = temp;
6  } // swap()
```

If you called `swap()` with two doubles instead, the compiler would use the template to generate the following code:

```
1  template <typename T>
2  void swap(T& a, T& b) {
3    T temp = a;
4    a = b;
5    b = temp;
6  } // swap()
```

```
1  // Compiler deduces that T is double
2  void swap(double& a, double& b) {
3    double temp = a;
4    a = b;
5    b = temp;
6  } // swap()
```

A function template can have more than one type parameter, as shown below.

```
1  template <typename T1, typename T2>
2  void print_two_types(T1& a, T2& b) {
3    std::cout << a << " " << b << std::endl;
4  } // print_two_types()
5
6  int main() {
7    std::string a = "EECS";
8    int32_t b = 281;
9    print_two_types(a, b);
10 } // main()
```

Here, the compiler deduces that `T1` is `std::string` and `T2` is `int32_t` from the arguments of the `print_two_types()` function call. It generates the code with the correct types using the template, and `"EECS 281"` is printed.

Templates can also be used when defining classes. To create a class template, create a class with ordinary member variable data types, and change them to type parameters in the same way as with function templates.

For example, suppose we wanted to create a `CustomPair` object that holds two objects of different types. Consider the following implementation, which only supports objects where both items are integers:

```
1  struct CustomPair {
2    int32_t first;
3    int32_t second;
4    CustomPair()
5      : first{}, second{} {}
6    CustomPair(int32_t first_in, int32_t second_in)
7      : first{first_in}, second{second_in} {}
8  };
```

If we wanted the `first` and `second` member variables to support any type, we can use a class template. To do so, simply add a template definition before the class definition, and replace the types of the member variables with type parameters.

```
1  template <typename T1, typename T2>
2  struct CustomPair {
3    T1 first;
4    T2 second;
5    CustomPair()
6      : first{}, second{} {}
7    CustomPair(T1 first_in, T2 second_in)
8      : first{first_in}, second{second_in} {}
9  };
```

When instantiating an instance of a templated object, the types of the template type parameters need to be specified! For example, suppose you wanted to initialize a `CustomPair`, and you attempted to initialize one like this:

```
CustomPair cp;
```

Since `CustomPair` is templated, there is no way for the compiler to deduce what `T1` and `T2` are! As a result, you will need to specify this information to create an instance of a `CustomPair`. To do so, add the correct types between angle brackets upon the object's declaration, in the same order as in the template definition. For instance, if the template definition is

```
template <typename T1, typename T2>
```

and you wanted to create a `CustomPair` where `T1` is `int32_t` and `T2` is `double`, you would declare the `CustomPair` as:

```
CustomPair<int32_t, double> cp;
```

By doing this, the compiler uses the provided template to generate the following definition of the `CustomPair` object, where `first` is of type `int32_t` and `second` is of type `double`.

```
struct CustomPair {
  int32_t first;
  double second;
  CustomPair()
    : first{}, second{} {}
  CustomPair(int32_t first_in, double second_in)
    : first{first_in}, second{second_in} {}
};
```

Class templates are extremely useful for container classes, as it allows you to define generic containers that support many different object types. For instance, when you declare a `std::vector<int>`, you are actually utilizing a class template to tell the vector that it should hold objects of type `int`. In fact, the C++ standard *template* library (STL) — which you will be very familiar with after completing this course — implements many common data structures and algorithms in a templated manner.

※ 1.9.2 *Variadic Templates* (⁎)

Variadic templates are templates that can take in a variable number of arguments. In C++, these templates are defined recursively: you process some number of arguments, and then you recurse with the remaining elements. You will also need to implement a "base case" templated function call that will be invoked once there are no more arguments to recurse on. An example of variadic template syntax is shown below:

```
1  template <typename T>
2  void print(T last_elt) {
3    std::cout << last_elt << std::endl;
4  } // print()
5
6  template <typename T, typename... Args>
7  void print(T first_elt, Args... rem_elts) { // parameter pack
8    std::cout << first_elt << " ";
9    print(rem_elts...); // parameter unpacking
10 } // print()
11
12 int main() {
13   print("a", 5, 1 + 2, "apple"); // prints "a 5 3 apple"
14 } // main()
```

Notice that we have two versions of the `print()` function: one that takes in a single argument (line 2) and one that takes in multiple arguments (line 7). If multiple arguments are passed into `print()`, the second version of `print()` will be used, with each call processing the arguments one at a time. For example, let us consider the outcome of `print("a", 5, 1 + 2, "apple")`.

```
1   template <typename T>
2   void print(T last_elt) {
3     std::cout << last_elt << std::endl;
4   } // print()
5
6   template <typename T, typename... Args>
7   void print(T first_elt, Args... rem_elts) { // parameter pack
8     std::cout << first_elt << " ";
9     print(rem_elts...); // parameter unpacking
10  } // print()
```

Multiple arguments are passed into `print()`, so the second function definition is chosen. The first argument (`"a"`) becomes `first_elt`, while the remaining elements (`5`, `1 + 2`, `"apple"`) are packed in the variable `rem_elts`. Line 8 would therefore print out `"a"`, and line 9 would recursively call `print()` using the remaining arguments of `5`, `1 + 2`, and `"apple"`.

```
5   // print("a", 5, 1 + 2, "apple")
6   template <typename T, typename... Args>
7   void print(T first_elt, Args... rem_elts) { // first_elt = "a", rem_elts = 5, 1 + 2, "apple"
8     std::cout << first_elt << " "; // prints out "a"
9     print(rem_elts...); // calls print(5, 1 + 2, "apple")
10  } // print()
```

Since there are still multiple arguments remaining, the second definition of `print()` is called again. The next argument to unpack, `5`, becomes `first_elt`, and the remaining elements of `1 + 2` and `"apple"` are stored in `rem_elts`. Line 8 prints out `5`, and line 9 recursively calls `print()` using the arguments of `1 + 2` and `"apple"`.

```
5   // print(5, 1 + 2, "apple")
6   template <typename T, typename... Args>
7   void print(T first_elt, Args... rem_elts) { // first_elt = 5, rem_elts = 1 + 2, "apple"
8     std::cout << first_elt << " "; // prints out 5
9     print(rem_elts...); // calls print(1 + 2, "apple")
10  } // print()
```

This process repeats again for the two remaining arguments. The next argument to unpack, `1 + 2`, becomes `first_elt`, and the remaining argument, `"apple"`, is stored in `rem_elts`. Line 8 prints out `3`, and line 9 recursively calls `print()` using the argument `"apple"`.

```
5   // print(1 + 2, "apple")
6   template <typename T, typename... Args>
7   void print(T first_elt, Args... rem_elts) { // first_elt = 3, rem_elts = "apple"
8     std::cout << first_elt << " "; // prints out 3
9     print(rem_elts...); // calls print("apple")
10  } // print()
```

Notice that there is only one argument remaining! As a result, the first definition of `print()` (on line 2) is chosen, and `"apple"` is passed in as the value of `last_elt`. Line 3 prints out `"apple"`, and our initial function call is complete. The final output is `"a 5 3 apple"`.

```
0   // print("apple")
1   template <typename T>
2   void print(T last_elt) { // last_elt = "apple"
3     std::cout << last_elt << std::endl; // prints "apple"
4   } // print()
```

The "base case" function definition is important! Without it, the compiler would not know what to do when there are no more arguments to unpack, and your code would not compile.

Variadic templated functions can also be used to process any number of arguments in a single call. The following is an example of a print function that prints out values in pairs:

```
1   template <typename T1, typename T2>
2   void print_pairs(T1 a, T2 b) {
3     std::cout << a << " " << b << std::endl;
4   } // print_pairs()
5
6   template <typename T1, typename T2, typename... Args>
7   void print_pairs(T1 first, T2 second, Args... args) {
8     std::cout << first << " " << second << std::endl;
9     print_pairs(args...);
10  } // print_pairs()
11
12  int main() {
13    print_pairs("a", 5, 1 + 2, "apple");
14  } // main()
```

The output of the above code is:

```
a 5
3 apple
```

However, if you do handle multiple arguments at once, you need to make sure that there is always a matching call for each possible base case. For instance, suppose we tried to call `print_pairs()` on an odd number of arguments:

```
template <typename T1, typename T2>
void print_pairs(T1 a, T2 b) {
  std::cout << a << " " << b << std::endl;
} // print_pairs()

template <typename T1, typename T2, typename... Args>
void print_pairs(T1 first, T2 second, Args... args) {
  std::cout << first << " " << second << std::endl;
  print_pairs(args...);
} // print_pairs()

int main() {
  print_pairs("a", 5, 1 + 2, "apple", "sauce");
} // main()
```

The above code would invoke the following versions of `print_pairs()` while unpacking the arguments:

```
print_pairs(std::string, int, <int, std::string, std::string>);  // second definition
print_pairs(int, std::string, <std::string>);  // second definition
print_pairs(std::string);  // no matching function call
```

There is no matching version of `print_pairs()` that takes in a single templated argument, so the third call would fail. The compiler would detect this and issue an error if you tried to compile this code.

Remark: As another example, below is a templated function that prints out the contents of a priority queue, which is a C++ standard library container that allows you to access the highest priority element at any point in time (do not worry about priority queues yet if you are not familiar with them; this remark will make more sense after you have covered them in class). Since priority queues can support many template parameters, you cannot template out just the element type if you want the `print_priority_queue()` method to accept any type of priority queue. Instead, you can use variadic templates to handle these additional template parameters, as shown:

```
template <typename Type, typename... Params>
void print_priority_queue(std::ostream& os, const std::priority_queue<Type, Params...>& pq) {
  std::priority_queue<Type, Params...> pq_to_print = pq;
  while (!pq_to_print.empty()) {
    os << pq_to_print.top() << " ";
    pq_to_print.pop();
  } // while
} // print_priority_queue()

int main() {
  std::vector<int32_t> nums = {281, 280, 376, 203, 183, 370};
  std::vector<std::string> strs = {"banana", "cherry", "apple"};

  std::priority_queue<int32_t> max_pq{nums.begin(), nums.end()};
  std::priority_queue<int32_t, std::vector<int32_t>, std::greater<int32_t>>
    min_pq{nums.begin(), nums.end()};
  std::priority_queue<std::string, std::deque<std::string>, std::greater<std::string>>
    str_pq{strs.begin(), strs.end()};

  // the templated print_priority_queue() method can be used on all three priority queues
  print_priority_queue(std::cout, max_pq);  // prints "376 370 281 280 203 183 "
  print_priority_queue(std::cout, min_pq);  // prints "183 203 280 281 370 376 "
  print_priority_queue(std::cout, str_pq);  // prints "apple banana cherry "
} // main()
```

1.10 Type Aliases

Type aliasing allows you to assign keywords to different types in C++. Using aliases can make programming easier and more efficient, especially when type names become extremely long.

Suppose you have a pair, where the `first` value stores a student's ID (as an `int32_t`) and the `second` value stores a pair object containing the first and last names of this student:

```
std::pair<int32_t, std::pair<std::string, std::string>>
```

This would be nasty to type out every time you wanted to create an object of this type! To address this, type aliasing allows you to give an alias to the type. For example, if you wanted to give the above type the alias `StudentPair`, you would be able to declare objects of this type with its alias instead. In other words,

```
std::pair<int32_t, std::pair<std::string, std::string>> my_student;
```

could instead be initialized using

```
StudentPair my_student;
```

where `StudentPair` is another name for `std::pair<int32_t, std::pair<std::string, std::string>>`.

To define an alias, simply use the keyword `using`, as follows:

```
using <ALIAS> = <ORIGINAL TYPE NAME>;
```

The following would create the alias `StudentPair` for the type `std::pair<int32_t, std::pair<std::string, std::string>>`.

```
using StudentPair = std::pair<int32_t, std::pair<std::string, std::string>>;
```

It is perfectly fine to assign multiple aliases to a single type. For instance, you could do this:

```
using Uniqname = std::string;
using FirstName = std::string;
using LastName = std::string;
std::pair<Uniqname, std::pair<FirstName, LastName>> myStudent = ...
```

In this case, `Uniqname`, `FirstName`, and `LastName` are all aliases for the `std::string` type.

1.11 Automatic Type Deduction

※ 1.11.1 *The `auto` Keyword*

Beginning with C++11, you can declare a variable without specifying its type if you use the keyword `auto` in place of the type. If `auto` is used to declare a variable, the type of the variable is deduced from its initialization value. Because of this, a variable declared with `auto` must also be initialized to a value. In the following example, the type of `j` is deduced from the initialization value of 281, which is an `int`.

```
int i = 281;    // i is int
auto j = 281;   // j is int
auto& k = i;    // k is int&
```

The `auto` keyword can be a double-edged sword. Using type deduction can make your code cleaner, but it can also make it harder to understand, especially if `auto` masks out a type that is not trivial to find (e.g., an ambiguous function return type defined in a distant section of code). In general, you should use type deduction to make your code clearer or safer, and not just to avoid the inconvenience of writing an explicit type.[3] Some types (such as iterator types in a loop) can be replaced with `auto` without issue. However, you should refrain from such behavior for important types that may provide useful information for others, if the reason for doing so is for mere personal convenience.

※ 1.11.2 *Deducing Function Return Types* (✻)

Suppose you have a templated function whose return types depend on its parameters, as shown below. What should the return type be?

```
template <typename T1, typename T2>
??? add(T1 a, T2 b) { return a + b; }
```

This is not apparent, since we do not know what the type of `T1 + T2` is. In C++11, we can address this problem by using the *trailing return type* syntax for defining a function, alongside the `decltype` keyword. As you should be aware, the standard convention for defining a function header is to specify the return type *before* the function name, as shown:

```
int32_t add(int32_t x, int32_t y) { return a + b; };
```

Using the trailing return type syntax, we can preface the function name with the `auto` keyword and then specify the return type *after* the function name and parameters:

```
auto add(int32_t x, int32_t y) -> int32_t { return a + b; };
```

Under this new syntax, we can elegantly express the return type of the function as follows:

```
template <typename T1, typename T2>
auto add(T1 a, T2 b) -> decltype(a + b) { return a + b; }
```

Here, `decltype` is a specifier that can be used to query the type of an expression, and it is commonly used to signify types that depend on templated parameters. In this case, `decltype(a + b)` represents the type of the expression `a + b`, which is used as the return type of the function. Note that you cannot simply use `decltype(a + b)` as the return type using the old syntax, since the argument names `a` and `b` would not be in scope (this is because the compiler parses the leading return type before the function parameters are declared).

```
// doesn't compile, compiler doesn't know what a and b are when evaluating the function return type
template <typename T1, typename T2>
decltype(a + b) add(T1 a, T2 b) { return a + b; }
```

Thus, if you are using C++11, the trailing return type syntax is the cleanest way to handle a templated function whose return type depends on its parameters (there is a way to do it without the trailing return type, but it is convoluted and not as elegant). However, C++14 introduced the `decltype(auto)` specifier, as well as automatic return type deduction without needing to explicitly specify the return type. By using `decltype(auto)`, the following code now compiles, and the return type of the templated function is automatically deduced:

```
// this compiles now
template <typename T1, typename T2>
decltype(auto) add(T1 a, T2 b) { return a + b; }
```

We will not go into too much detail on how `decltype` works here, but you are definitely encouraged to read more about it on your own.

[3] From Google's C++ style guide: https://google.github.io/styleguide/cppguide.html#Type_deduction

1.12 Prefix and Postfix Incrementation

In C++, you can increment or decrement the value of a variable by using `operator++` or `operator--`. However, it matters whether you put the operator *before* a variable name (`++i`) or *after* a variable name (`i++`). Putting `++` in front of a variable (prefix incrementation) will increment the value of the variable and then return the incremented value.

```
int32_t i = 281;
int32_t j = ++i;                // increment i, then store in j
std::cout << j << std::endl;    // prints 282
```

Putting `++` after a variable (postfix incrementation) will increment the value of the variable, but return the original value of the variable *before it was incremented*. You can essentially treat this as if the incremention is done at the *very end*.

```
int32_t i = 281;
int32_t j = i++;                // store in j, then increment i
std::cout << j << std::endl;    // prints 281
```

The same applies for decrementation using `operator--`.

Is one method better than the other? The details are a bit nuanced, so you don't need to know the full details for this course. However, if you do not need to use the previous value before a variable is incremented or decremented, it is generally good practice to prefer prefix (`++i`) over postfix (`i++`). Although many compilers can optimize away differences in many cases, this is not always guaranteed, and in cases where both variations work equally, the prefix form is never less efficient than the postfix form.

1.13 Loops

※ 1.13.1 *For and While Loops*

At this point, you should probably be familiar with loops in C++, particularly `for` loops and `while` loops with the following usage:

```
for (init; condition; update) {
  // code
}

while (condition) {
  // code
}
```

In the `for` loop, the `init` statement is executed once before the execution of the code inside the loop, the `condition` statement identifies when the loop should be run, and the `update` statement is executed every time the code in the loop has been executed. Once the `condition` statement is no longer true, the loop terminates. Similar logic can be applied to the `while` loop, which loops through a block of code as long as its corresponding condition is true.

You can always convert between a `for` loop and a `while` loop, and vice versa. To convert a `for` loop into a `while` loop, add the initialization statement before the `while` loop and the update statement to the end of the `while` loop's code block.

```
// for loop
for (int32_t i = 0; i < 10; ++i) {
  // code
} // for i

// converted while loop
int32_t i = 0;
while (i < 10) {
  // code
  ++i;
} // while
```

To convert a `while` loop to a `for` loop, you can omit the initialization statement.

```
// while loop
while (*ptr++ != nullptr) {
  // code
  ++counter;
} // while

// converted for loop
for(; *ptr != nullptr; ++ptr, ++counter) {
  // code
} // for
```

In the following sections, we will discuss two additional types of loops that you may encounter in this class.

※ 1.13.2 *Range-Based For Loop*

If you want to loop through all elements in a given collection of elements, you can use the range-based `for` loop. The range-based `for` loop follows the following format:

```
for (individual element in collection : collection) {
  // do stuff here
}
```

For example, the following two loops do the exact same thing:

```
std::vector<int32_t> vec = {1, 2, 3, 4, 5};

// traditional for loop, uses index counter to visit each element
for (size_t i = 0; i < vec.size(); ++i) {
  vec[i] *= 2;
} // for i

// range-based for loop, visits every element directly and stores value in x
for (int32_t& x : vec) {
  x *= 2;
} // for x

// range-based for loop using auto, does the same thing as above
for (auto& x : vec) {
  x *= 2;
} // for x
```

In the parentheses of the range-based `for` loop, there are two things you need: a variable that stores the current element you are visiting, and the collection of elements you want to iterate over. These terms are separated with a colon. For instance, `for (int& x : vec)` loops through every element in the container `vec`, where `x` is the element that the loop is currently visiting. In the example where `vec = {1, 2, 3, 4, 5}`, the value of `x` is `1` on the first iteration of the range-based `for` loop, `2` on the second iteration, `3` on the third iteration, and so on.

If you are using the range-based `for` loop to modify something in a container, make sure to declare the iteration variable as a reference (`&`)! If you don't declare this variable as a reference, the range-based `for` loop *makes a copy* of each element in the search container and stores the copy in the iteration variable. If line 9 did not have a reference, line 10 would modify a copy of each element in `vec`, rather than the actual element in `vec` itself. This would leave the elements in `vec` unchanged, even after the loop completes).

※ 1.13.3 *Do While Loop*

The *do while loop* is a post-test loop. The body of a do while loop always runs at least once, with the condition check of the test expression at the end. If the test expression is true, the body of the loop is executed again; otherwise, the do while loop is terminated. The do while loop follows the following syntax:

```
do {
  // body of loop
} while (condition);
```

As an example, consider the following code:

```
int32_t num = 281;
do {
  std::cout << "EECS " << num << std::endl;
  ++num;
} while (num < 280);
```

Notice how the value of `num` is initialized to `281`, so the condition of `num < 280` is never true! However, since the do while loop is a post-test loop, the body of the loop runs is guaranteed to run *at least once*, regardless of whether the condition is true or not. In this case, the program prints out `"EECS 281"` on its first iteration of the loop. Only after this first iteration is complete does the program check to see if the `while` condition is met — in this case, it isn't, so the loop terminates after the first iteration.

※ 1.13.4 *The `break` Keyword*

Two important keywords involved with loops are `break` and `continue`. When a `break` statement is reached in a program, it terminates the smallest enclosing loop. Execution of the program resumes immediately after the body of the terminated loop. This is illustrated below:

```
while (/* condition */) {
  // do stuff
  if (/* condition for break */) {
    break;
  } // if
  // do more stuff
} // while

// more code here
```

When the `break` statement is reached, the while loop immediately terminates.

Encountering a `break` statement causes a loop to finish, even if its end condition is now fulfilled. Thus, `break` can be used to end an otherwise infinite loop, or to terminate a loop before its natural end.

Note that, if `break` is encountered in a nested loop, you would only exit from the smallest enclosing loop that you are in. In the following code, for example, the `break` on line 4 causes the program to exit the `for` loop defined on line 2, but not the `while` loop defined on line 1.

```
1  while (true) {
2    for (int32_t i = 0; i < 5; ++i) {
3      if (i == 3) {
4        break;
5      } // if
6      std::cout << i << " ";
7    } // for i
8    std::cout << "PotatoBot" << std::endl;
9  } // while
```

As a result, the output of "`0 1 2 PotatoBot`" would be printed indefinitely in this example.

※ 1.13.5 *The `continue` Keyword*

The `continue` statement is very similar to `break`. However, instead of terminating the loop, a `continue` statement forces the next iteration of the loop to take place, skipping any remaining code in the current iteration. This is illustrated below:

```
1  while (/* condition */) {
2    // do stuff
3    if (/* condition for continue */) {
4      continue;
5    } // if
6    // do more stuff
7  } // while
```

When the `continue` statement is reached, the loop skips to the next iteration, and the remaining code in the current iteration does not run.

An example of code that uses `continue` is shown below. This code prints out every integer from 1 to 5, except for 3.

```
1  for (int32_t i = 1; i <= 5; ++i) {
2    if (i == 3) {
3      continue;
4    } // if
5    std::cout << i << " ";
6  } // for i
```

1.14 Ternary Operator

The *ternary (conditional) operator* in C++ (denoted by a question mark character) provides an alternative way to express a conditional statement. The syntax of a ternary operator is as follows:

```
<CONDITION> ? <EXPRESSION IF TRUE> : <EXPRESSION IF FALSE>
```

The ternary operator can be used to shorten a standard if/else statement into one line. For example, consider the following function, which returns whether a student received a passing or failing grade.

```
1  bool passed_class(int32_t score) {
2    if (score >= 70) {
3      return true;
4    } // if
5    else {
6      return false;
7    } // else
8  } // passed_class()
```

With a ternary operator, we can condense the if/else onto a single line.

```
1  bool passed_class(int32_t score) {
2    return score >= 70 ? true : false;
3  } // passed_class()
```

Here, the statement before the `?` dictates whether we return the value before the colon (`true`) or the value after the colon (`false`). If the given score is ≥ 70, then the function returns `true`; otherwise, it returns `false`. Another example is shown below:

```
1  bool try_buy_item(double price, double willingness_to_pay, double& total_savings) {
2    double savings_from_purchase;
3    if (price < willingness_to_pay) {
4      savings_from_purchase = willingness_to_pay - price;
5    } // if
6    else {
7      savings_from_purchase = 0.0;
8    } // else
9    total_savings += savings_from_purchase;
10   return price <= willingness_to_pay;
11 } // buy_item()
```

Using a ternary operator, this same function can be written as follows:

```
1  bool try_buy_item(double price, double willingness_to_pay, double& total_savings) {
2    double savings_from_purchase = price < willingness_to_pay ? willingness_to_pay - price : 0.0;
3    total_savings += savings_from_purchase;
4    return price <= willingness_to_pay;
5  } // buy_item()
```

1.15 Inheritance and Polymorphism

※ 1.15.1 *Inheritance*

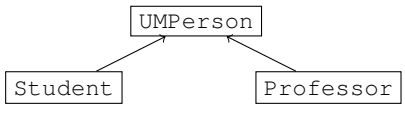
Object-oriented programming (OOP) is a computer programming approach defined by solving complex problems using objects and their methods. There are four main principles to object-oriented programming: encapsulation, abstraction, inheritance, and polymorphism.

We covered the first two of these principles in earlier sections. **Encapsulation** is the act of combining both data and methods into a single unit. This can be done using a `class` in C++, where the state of an object is stored as private member variables, and methods that modify this internal state are provided as public member functions. **Abstraction** is the idea that internal implementation details should be hidden from the user, and that only relevant operations for usage should be revealed. For example, if you want to increase the speed of a car, you just need to know to press the accelerator; you do not need to know the underlying details of how the car works! Similarly, if you want to remove an element from a `std::vector<>`, you just need to call the `std::vector::erase()` member function; you do not need to understand how this member function is implemented under the hood.

In this section, we will cover the concepts of inheritance and polymorphism. **Inheritance** is the ability for a class to reuse the interface or functionality of another class. The need for inheritance arises out of the fact that many objects in OOP design are very similar but not entirely the same. Inheritance provides a means for similar objects to share common logic while also exhibiting logic that is individually distinct. Let's look at an example: suppose we define the following `class`, which represents a person at U-M. Every person at U-M has a name, a UMID, and an age. Since these member variables are private, the public getting functions starting on line 9 allow us to retrieve their values.

```
1   class UMPerson {
2     std::string name;
3     int32_t id;
4     int32_t age;
5   public:
6     UMPerson(std::string name_in, int32_t id_in, int32_t age_in)
7       : name{name_in}, id{id_in}, age{age_in} {}
8
9     std::string get_name() const {
10      return name;
11    } // get_name()
12
13    int32_t get_id() const {
14      return id;
15    } // get_id()
16
17    int32_t get_age() const {
18      return age;
19    } // get_age()
20  };
```

However, not everyone at the university has the same traits! We can further break the `UMPerson` type down into subtypes, such as students and professors. Both students and professors are considered as a `UMPerson`, and both groups have names, UMIDs, and ages. However, these groups are also distinct from each other in several ways. For example, students have a GPA, a grade level, a list of classes taken, and a graduation year. On the other hand, professors have information on a list of classes taught, the number of years they have been at the university, a salary, and a rank or title. With inheritance, we can define both a `Student` and `Professor` object that "inherit" shared traits from the `UMPerson` class but also behave slightly differently on their own.



The class that contains the common functionality, in this case `UMPerson`, is known as a *base class*. We can then make subclasses from the base class that either change the functionality in subtle ways or introduce new functionality altogether — these subclasses are known as *derived classes*. In this example, the `Student` and `Professor` classes are derived from the base class `UMPerson`. To create a derived class, add a colon after the name of the derived class upon declaration, then the keyword `public`, then the name of the base class. The following defines a class named `Student` that derives from the base class `UMPerson`:

```
1   class Student : public UMPerson {
2     std::vector<std::string> classes_taken;
3     double gpa;
4     int32_t grade_level;
5     int32_t graduation_year;
6   };
```

By doing this, the `Student` class has everything that the `UMPerson` class has! Notice that we used the `public` keyword when defining our derived class. This allows our derived `Student` class to directly access any *non-private* member functions or variables defined in the base `UMPerson` class (known as *public inheritance*).[4] Note that members *private* to `UMPerson` are inherited but *cannot be directly accessed* by the derived `Student` class, which is why we needed the getter functions in the `UMPerson` interface.

[4]There are other ways to inherit from a base class, such as private or protected inheritance, which can be used to change the level of access for non-private inherited members. There alternative approaches, however, are not as common.

In order to properly initialize a derived class, its constructor must properly initialize the base class it derives from. In our example, not only do we need to initialize the `gpa`, `grade_level`, and `graduation_year` member variables upon creating a `Student`, we also need to initialize the `name`, `id`, and `age` member variables that were inherited from `UMPerson`. Since these three inherited members are private in the `UMPerson` class, the `Student` class cannot initialize these member variables directly. Instead, the `Student` constructor initializes these three members by invoking the base class (`UMPerson`) constructor in the member-initializer list of its own constructor, as shown:

```
Student(std::string name_in, int32_t id_in, int32_t age_in,
        double gpa_in, int32_t grade_in, int32_t year_in)
  : UMPerson{name_in, id_in, age_in}, gpa{gpa_in}, // init inherited members using base class ctor
    grade_level{grade_in}, graduation_year{year_in} {}
```

The constructor of a derived class will always invoke a constructor of the base class. If the constructor of the base class is not explicitly invoked in the member-initializer list of the derived class, a call to the base class's default constructor is made implicitly.

The following `Professor` class follows the same vein as the `Student` class; it inherits the `name`, `age`, and `id` member variables from the `UMPerson` class (since these variables are shared across all people at U-M), while having its own member variables unique to the `Professor` class, such as `salary` and `department`:

```
class UMPerson {
  std::string name;
  int32_t id;
  int32_t age;
public:
  UMPerson(std::string name_in, int32_t id_in, int32_t age_in)
    : name{name_in}, id{id_in}, age{age_in} {}
  std::string get_name() const { return name; }
  int32_t get_id() const { return id; }
  int32_t get_age() const { return age; }
};

class Professor : public UMPerson {
  std::vector<std::string> classes_taught;
  std::string department;
  std::string title;
  double salary;
  int32_t years_taught;
  bool tenured;
public:
  Professor(std::string name_in, int32_t id_in, int32_t age_in, std::string dep_in,
            std::string title_in, double salary_in, int32_t years_in, bool tenured_in)
    :  UMPerson{name_in, id_in, age_in}, department{dep_in}, title{title_in},
       salary{salary_in}, years_taught{years_in}, tenured{tenured_in} {}
  // other member functions specific to a Professor
  ...
};
```

※ 1.15.2 *Polymorphism*

The fourth and final principle of object-oriented programming is **polymorphism**, which allows a derived class object to be used whenever a base class object is expected.[5] In C++, you are allowed to use a reference or pointer of a base type to refer to an object of a derived type that publicly inherits from it. For instance, assume we had a `Student` object named `my_student` and a `Professor` object named `my_professor`. Since both classes are derived from the base class `UMPerson`, we can use a `UMPerson` pointer (`UMPerson*`) or a `UMPerson` reference (`UMPerson&`) to refer to our `Student` and `Professor`.

```
UMPerson* student_ptr = &my_student;     // okay, since Student is a UMPerson
UMPerson& student_ref = my_student;      // okay, since Student is a UMPerson
UMPerson* prof_ptr = &my_professor;      // okay, since Professor is a UMPerson
UMPerson& prof_ref = my_professor;       // okay, since Professor is a UMPerson
```

The other way around is not implicitly allowed; you cannot use a reference or pointer of a derived type to refer to an object of the base type.

```
UMPerson myUMP{"Bob", 12345678, 21};     // create a U-M Person
Student* student_ptr = &myUMP;           // not okay!
Student& student_ptr = myUMP;            // not okay!
Professor* prof_ptr = &myUMP;            // not okay!
Professor& prof_ptr = myUMP;             // not okay!
```

All students are people at U-M, so it would be safe to use a `UMPerson*` or `UMPerson&` to refer to a `Student`. However, not all people at U-M are students, so it would be unsafe to convert the other way around! As a result, if you want to cast a base class to a derived class, you must do so explicitly using `static_cast<>` or `dynamic_cast<>` (which will be covered in more detail in the next section).

[5] There are several types of polymorphism, but this definition refers to *subtype polymorphism*, which is what the term "polymorphism" generally usually refers to.

It should also be mentioned that, although we can use a pointer or reference of a base class to refer to a derived class, the following would *not* be okay. This is because, even though an object of type `Student` derives from an object of type `UMPerson`, the value of a `Student` *does not necessarily* fit into a `UMPerson` object.

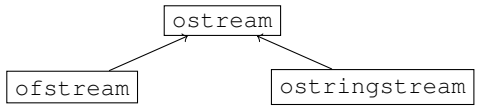
```
UMPerson myUMP = myStudent;
```

In this case, trying to fit a `Student` object into an object of type `UMPerson` would result in *object slicing* — only the members defined by `UMPerson` would be copied, and the other member variables defined in `Student` but not in `UMPerson` would be "sliced off."

Polymorphism allows us to pass a derived-class object into a function that expects a base-class object. There are many places where this behavior can be helpful. As an example, let's revisit the overloaded `operator<<` that we wrote earlier.

```
1  std::ostream& operator<<(std::ostream& os, const Point& pt) {
2    // define the output format you want to print for this custom object
3    os << "(" << pt.x << ", " << pt.y << ")";
4    return os;
5  } // operator<<()
```

Notice that the `operator<<` overload expects an object of type `ostream` as its first argument. However, the `ostream` type is actually a base class for several other output stream types, as shown below:



Because of this relationship, polymorphism allows us to use `operator<<` on an object of type `ostringstream` (the derived class) even though an object of type `ostream` (the base class) is expected.

```
1  int main() {
2    Point pt{3, 4};
3    std::ostringstream os;
4    os << pt;                              // operator<< works on ostringstream
5    std::cout << os.str() << std::endl;    // prints (3, 4)
6  } // main()
```

Polymorphism is also helpful if you want to create a container of base class pointers, where each pointer points to a different derived class object. As an example, let's use a simplified version of the `Student` and `Professor` classes covered previously.

```
1  class UMPerson {
2  public:
3    std::string name;
4    UMPerson(std::string name_in) : name{name_in} {}
5    void print_name() const { std::cout << "Person " << name << std::endl; }
6  };
7
8  class Student : public UMPerson {
9  public:
10   Student(std::string name_in) : UMPerson{name_in} {}
11   void print_name() const { std::cout << "Student " << name << std::endl; }
12 };
13
14 class Professor : public UMPerson {
15 public:
16   Professor(std::string name_in) : UMPerson{name_in} {}
17   void print_name() const { std::cout << "Professor " << name << std::endl; }
18 };
```

Let's initialize several students and professors and add them all to a vector of `UMPerson` pointers. Then, let's go through each object in our vector and call its `print_name()` member function.

```
1  int main() {
2    Student a = Student{"Alice"};
3    Student b = Student{"Bob"};
4    Professor c = Professor{"Cathy"};
5    Professor d = Professor{"Dave"};
6    std::vector<UMPerson*> vec = { &a, &b, &c, &d };
7    for (const auto& person : vec) {
8      person->print_name(); // call print_name() of each object in vector
9    } // for
10 } // main()
```

Running this code ends up producing the following output.

```
Person Alice
Person Bob
Person Cathy
Person Dave
```

By referring to all of the derived classes in the vector as a pointer of the base class, we ended up using the base class's `print_name()` function for every object as well! Although Alice is a `Student`, calling Alice's `print_name()` function runs the `print_name()` function defined in the `UMPerson` class, rather than the one defined in the `Student` class.

To fix this, we will use the `virtual` keyword. When a `virtual` function is called, the compiler determines the correct version of the function to call based on the type of the object *during runtime* (a process known as *dynamic binding*) rather than the type of the object at compile time (which is known as *static binding*). The `virtual` keyword is only necessary in the base class; it is optional for derived classes. Furthermore, it can only be prepended to function definitions within a class — if the function is defined outside of the class, the `virtual` keyword need not be applied.

As mentioned, the `virtual` keyword should be used for the base class. For derived classes, you should use the `override` keyword, which lets the compiler know that the function intends to override a member function in the base class. This keyword can help detect bugs, as the compiler would produce an error if a function declared with `override` does not end up overriding anything. The following code is the result of adding the `virtual` and `override` keywords to our original member functions.

```
1  class UMPerson {
2  public:
3    std::string name;
4    UMPerson(std::string name_in) : name{name_in} {}
5    virtual void print_name() const { std::cout << "Person " << name << std::endl; }
6  };
7
8  class Student : public UMPerson {
9  public:
10   Student(std::string name_in) : UMPerson{name_in} {}
11   void print_name() const override { std::cout << "Student " << name << std::endl; }
12 };
13
14 class Professor : public UMPerson {
15 public:
16   Professor(std::string name_in) : UMPerson{name_in} {}
17   void print_name() const override { std::cout << "Professor " << name << std::endl; }
18 };
```

If we run our code again with the `virtual` and `override` keywords in place, we get the following output:

```
Student Alice
Student Bob
Professor Cathy
Professor Dave
```

This is exactly what we want, as the version of `print_name()` called matches the type of the object that calls it.

1.16 Casting

Casting is the process of converting data from one type to another. In C++, there are two types of conversions: implicit and explicit conversions. As mentioned earlier, implicit conversions are performed automatically by the compiler, without input from the programmer.

```
int32_t a = 281;
double b = a;     // int implicitly converted to double
```

On the other hand, explicit conversions are explicitly specified by the programmer. In C++, there are four main types of explicit casts:

- `static_cast<>`
- `dynamic_cast<>`
- `reinterpret_cast<>`
- `const_cast<>`

※ 1.16.1 *Static Cast*

A *static cast* is the standard, "well-behaved" cast that can be used to convert between two compatible types. This is the preferred method of casting for automatic conversions and narrowing conversions (i.e., converting from a type with a wider range to a smaller range, such as `double` to `int`). *A static cast should be the preferred method to use if you want to cast a variable to a different type, provided that it is possible!* Not only are static casts more readable and express clearer intent, they are also safe: if you attempt to perform an incompatible conversion using `static_cast<>`, the compiler would issue an error.

Valid static cast:

```
double a = 281.99;
int32_t b = static_cast<int32_t>(a);  // casts a from double to int32_t, b is now 281
                                      // this is a narrowing conversion, which truncates decimals
```

Invalid static cast:

```
class MyClass {
  char a = 'a';
} mc;

char mc_a = static_cast<char>(mc);    // attempt to convert MyClass object to char, compile error
```

※ 1.16.2 *Dynamic Cast* (⁂)

A dynamic cast can be used to perform type-safe downcasting (e.g., converting a base class pointer/reference to a derived class pointer/reference). When you attempt a dynamic cast, the program actually performs a check *during runtime* to determine if the requested conversion is possible. If a dynamic cast fails, a `nullptr` is returned. In general, this cast is really only applicable if you are trying to downcast a polymorphic type. An example is shown below, using the `UMPerson` example from the previous section.

```
void process_person(UMPerson& person) {
  if (auto as_student = dynamic_cast<Student*>(&person)) {
    std::cout << "Received a student named " << as_student->name << std::endl;
  } // if
  else if (auto as_professor = dynamic_cast<Professor*>(&person)) {
    std::cout << "Received a professor named " << as_professor->name << std::endl;
  } // else if
} // process_person()

int main() {
  Student a{"Alice"};
  Professor b{"Bob"};
  process_person(a);   // prints "Received a student named Alice"
  process_person(b);   // prints "Received a professor named Bob"
} // main()
```

※ 1.16.3 *Reinterpret Cast* (⁂)

A reinterpret cast takes the memory representation of an object and reinterprets it as if it were the type you are trying to cast to. In other words, if you attempt to reinterpret cast an object, the underlying bits that represent the object in memory are treated as if they were of another type. Examples of reinterpret casts are shown below (you do not have to understand what is happening here):

```
struct MyClass {
  int16_t a;
  int16_t b;
};

int main() {
  double d = 281.37;
  uint64_t ptr_addr = reinterpret_cast<uint64_t>(&d);  // casts pointer to integer
  std::cout << ptr_addr << std::endl;                  // prints out int representation of address

  int32_t num = 24248601;
  MyClass* mc = reinterpret_cast<MyClass*>(&num);      // casts int mem representation to MyClass
  std::cout << mc->a << std::endl;                     // prints 281 ...!? (depending on system)
  std::cout << mc->b << std::endl;                     // prints 370 ...!? (depending on system)
} // main()
```

Because `reinterpret_cast<>` actually reinterprets physical bits in memory, it is a very dangerous cast that can cause misbehavior if you do not know what you are doing. This cast is only mentioned here for completeness on the topic, but you really should not do this in your code. If you ever need to make such a cast in the future, you will know (and it probably won't be anytime soon, if ever).

※ 1.16.4 *Const Cast* (⁂)

Joining `reinterpret_cast<>` as a cast that you should not use in this class, we have `const_cast<>`. A const cast can be used to remove the constness from a pointer or reference that refers to an object that is not const. An example is shown below:

```
int main() {
  int32_t i = 0;
  const int32_t& i_ref = i;
  // i_ref = 281;                       // not allowed, since i_ref is const
  const_cast<int32_t&>(i_ref) = 281;    // const cast i_ref and assign it to 281
  std::cout << i_ref << std::endl;      // prints 281
  std::cout << i << std::endl;          // prints 281
  // i_ref = 370;                       // not allowed, i_ref is still const

  int32_t j = 0;
  const int32_t* j_ptr = &j;
  // *j_ptr = 281;                      // not allowed, since j_ptr is const
  *const_cast<int32_t*>(j_ptr) = 281;   // const cast j_ptr and assign its pointer value to 281
  std::cout << *j_ptr << std::endl;     // prints 281
  std::cout << j << std::endl;          // prints 281
  // *j_ptr = 370;                      // not allowed, j_ptr is still const
} // main()
```

As before, this cast is only introduced here for completeness. This is not a cast that you should be frequently using, as it is not only dangerous if used incorrectly, but it is also indicative of poorly designed code.

※ 1.16.5 *C-Style Casts* (✻)

Apart from static casts, another cast that you may see often is the standard C-style cast. To perform a C-style cast, simply add the type you want to convert to in parentheses before the object you want to cast. An example is shown below:

```
double a = 281.99;
int32_t b = (int32_t)a;  // casts a from double to int32_t and assigns to b, b is now 281
```

However, you should avoid C-style casts when developing in C++. If you want to perform a cast, you should generally use `static_cast<>` instead. This is because a C-style cast can behave like a static cast, a const cast, or even a reinterpret cast depending on the types you are trying to convert. This can lead to potentially dangerous or undefined behavior if you mistakenly try to cast the wrong type. With `static_cast<>`, the compiler will detect incompatible conversions and issue an error during compile time, making it a safer way to cast different types. Usage of `static_cast<>` is also more readable and makes it easier to identify places in your code where type conversions are occurring.

1.17 Exception Handling

An **exception** is an event that is issued during a program's execution when erroneous conditions are encountered during runtime. Exceptions are often used to handle runtime errors in a program, and they provide several advantages over other traditional error handling mechanisms:

- *Exceptions allow error handling code and normal program code to be organized separately.* If you use exceptions to handle errors, you can structure your program so that your error handling is implemented elsewhere (instead of in the middle of your normal program code). This improves code readability and maintainability.
- *Exceptions are versatile, and different methods can pick and choose which exceptions it should handle.* A method can throw many exceptions, but it can choose which ones it wants to handle. If an exception is not handled, it is able to propagate upward to another caller until it finds something that is able to handle it.
- *Exceptions can be grouped together.* Thanks to inheritance, exception objects can be grouped into a hierarchy based on its type. This makes it easier to categorize different types of behaviors and allows similar errors to be handled together.

In C++, exceptions are handled using three keywords: `try`, `catch`, and `throw`. The `throw` keyword can be used to trigger an error handling event (we call this process "throwing an exception"). The code itself is segmented into `try` blocks (which include code that could potentially throw an exception) and `catch` blocks (which include code to handle exceptions that are thrown). Try-catch blocks are placed around code that could potentially encounter an error, as shown below:

```
1  int main() {
2    try {
3      // code that could potentially throw an exception
4    }
5    catch (const EXCEPTION_NAME& ex) {
6      // code that handles exceptions thrown within try block
7    }
8  }
```

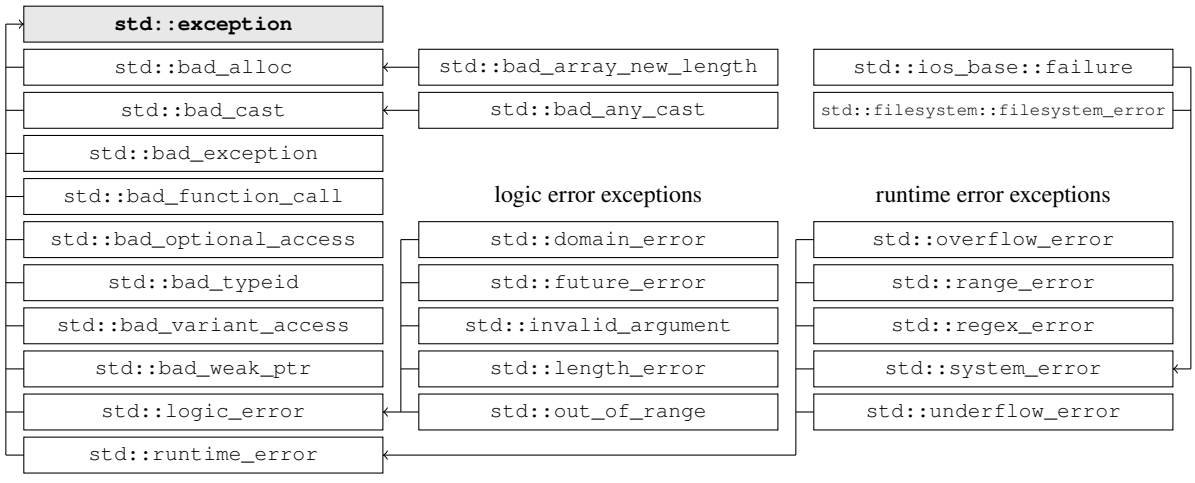
Each `try` block can be followed by multiple `catch` blocks that each handle a different type of exception. When an exception is thrown within a `try` block, the program looks at all of its associated `catch` blocks (from top to bottom), and program execution is transferred to the first `catch` block that is able to handle the exception that was thrown. After the code in this `catch` block runs to completion, execution proceeds beyond the entire `try-catch` block, and any remaining code that was not run in the `try` block is skipped.

If there exists no `catch` block that can handle an exception, the exception propagates to the caller of the function that threw the exception, and the program again attempts to look for a `catch` block that can potentially handle the exception. An uncaught exception continues propagating to the caller as long as there is no `catch` block that can handle it; if it ever propagates beyond `main()` without being handled, the program terminates, and the user is informed that an error occurred.

Technically, anything can be thrown from a `try` block. The following code throws the integer `fav_class` if it is not equal to 281:

```
1  int main() {
2    try {
3      int32_t fav_class;
4      std::cout << "What is your favorite class?" << std::endl;
5      std::cin >> fav_class; // reads in fav_class from user input
6      if (fav_class == 281) {
7        std::cout << "Of course! Who doesn't love EECS 281?" << std::endl;
8      } // if
9      else {
10       throw (fav_class);
11     } // else
12   } // try
13   catch (int32_t wrong_class) {
14     std::cout << "You said your favorite class is " << wrong_class << std::endl;
15     std::cout << "YOU ARE WRONG" << std::endl;
16   } // catch
17 } // main()
```

In this section, however, we will primarily focus on exceptions that inherit from the C++ standard library's `std::exception` class. These exceptions are defined using the following inheritance hierarchy (as of C++17):



Every exception in the language is derived from the base `std::exception` class, which implements a member function called `what()`. The `what()` member function returns a message that identifies the exception.

```
int main() {
  try {
    throw std::runtime_error{"descriptive message here"};
  } // try
  catch (const std::runtime_error& ex) {
    std::cerr << ex.what() << std::endl; // prints "descriptive message here"
  } // catch
} // main()
```

You do not need to remember all these exceptions, but there are a few that you are likely to see again. One of these is the `std::bad_alloc` exception, which is thrown whenever the `new` keyword fails to allocate the memory that was requested:

```
int main() {
  try {
    char* arr = new char[STACK_SIZE]; // throws bad_alloc if not enough memory
  } // try
  catch (const std::bad_alloc& ex) {
    std::cerr << "bad_alloc caught: " << ex.what() << std::endl;
  } // catch
} // main()
```

In addition, you can define custom exceptions that inherit from the `std::exception` base class. An example is shown below. Note that the `what()` member function is virtual, so you can override its implementation to return any message you want.

```
class EECS281Exception : public std::exception {
  std::string msg;
public:
  EECS281Exception(const std::string& msg_in)
    : msg{msg_in} {}

  const char* what() const noexcept override {
    return msg.c_str();
  } // what()
};

int main() {
  try {
    int32_t fav_class;
    std::cout << "What is your favorite class?" << std::endl;
    std::cin >> fav_class; // reads in fav_class from user input
    if (fav_class == 281) {
      std::cout << "Of course! Who doesn't love EECS 281?" << std::endl;
    } // if
    else {
      throw EECS281Exception{"YOU ARE WRONG"};
    } // else
  } // try
  catch (const EECS281Exception& ex) {
    std::cout << e.what() << std::endl; // prints "YOU ARE WRONG"
  } // catch
} // main()
```

Remark: On line 7 of the previous code, the `what()` function was declared using the `noexcept` keyword. What does this mean? The `noexcept` keyword is applied to functions that promise not to throw any exceptions. In other words, a `noexcept` function should never throw an exception on its own, nor should it use `new` to allocate memory, call another library function that could potentially throw an exception, perform arithmetic that could cause an overflow or underflow, or exhibit any other behavior that could cause an exception to be thrown. If something unexpected happens and the function somehow does throw an exception, the normal try-catch procedure is ignored and the program is terminated immediately.

Lastly, since `catch` clauses are checked in the order in which they appear (i.e., the first `catch` block that is able to handle an exception always runs), `catch` blocks should be ordered so that any derived exception type is handled *before* its base exception type. For example, if you have a `catch` block that specifically handles the `std::out_of_range` exception, and another `catch` block that handles all other logic errors, the block that catches the `std::out_of_range` exception should be listed before the block that catches the generic `std::logic_error`.

```
1  try {
2    /* ... code that could throw any type of logic error ... */
3  }
4  catch (const std::out_of_range& ex) {
5    /* ... handles std::out_of_range exception ... */
6  }
7  catch (const std::logic_error& ex) {
8    /* ... handles std::logic_error exception ... */
9  }
```

If you switch the order of these two `catch` statements, the `catch` block that handles `std::logic_error` exceptions would also end up handling `std::out_of_range` exceptions (due to polymorphism, since `std::out_of_range` is derived from `std::logic_error`). As a result, the catch block that handles `std::out_of_range` exceptions would never run at all.

It should also be noted that an ellipsis (…) can be used as a parameter of a `catch` statement, which enables the corresponding `catch` block to handle *any* object that is thrown. Since this block catches everything, it should be placed below all other `catch` blocks if included.

```
1  try {
2    /* ... code that could throw anything ... */
3  }
4  catch (const std::logic_error& ex) {
5    /* ... handles logic errors ... */
6  }
7  catch (const std::runtime_error& ex) {
8    /* ... handles runtime errors ... */
9  }
10 catch (...) {
11   /* ... handles anything else ... */
12 }
```

Exceptions are not a primary focus of this class, and you will not need to use them for your assignments. However, knowledge of how to handle exceptions may be important for future upper-level courses (and is a valuable skill overall). Even if you never throw any exceptions on your own in this class, you may still encounter them when you submit your code to the EECS 281 autograder! In the following autograder output, a test case failed because it went over the maximum allowable memory limit, forcing the program to throw a `std::bad_alloc` exception.