CSE 374: Programming Concepts and Tools

Spring 2026
Instructor: Megan Hazen

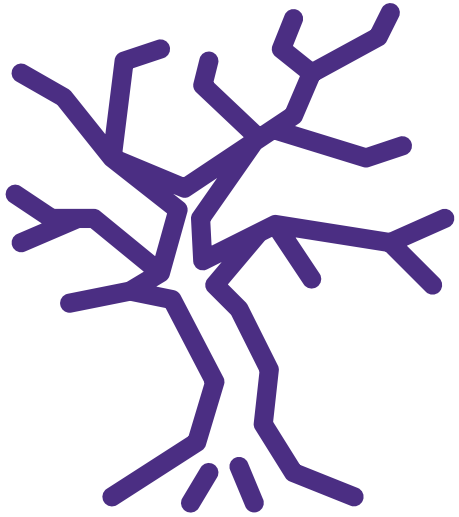
Today's Goals

Subject Matter

- C++ Templates
- Standard Template Library

Your Goals

- Practice Exercises
- Finish HW6
- HW7
- Did you do your debugging demo?



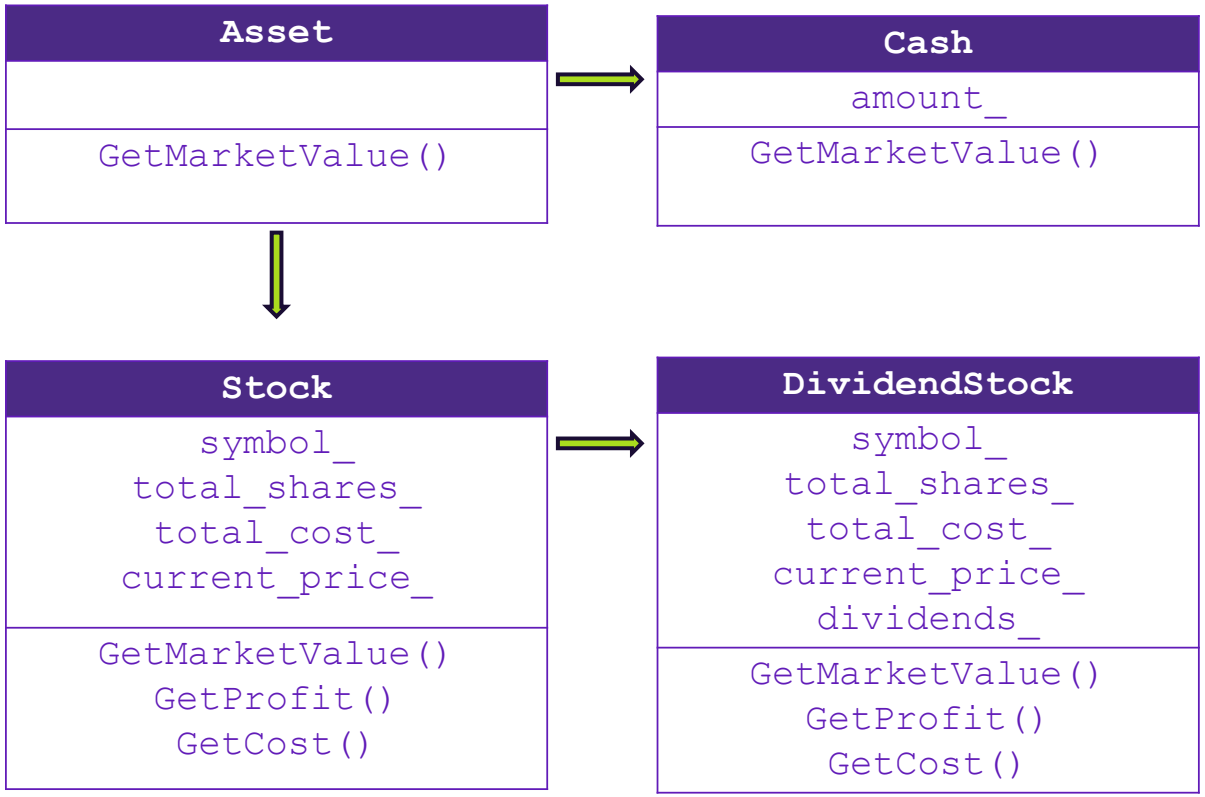
INHERITANCE

OOP fundamentals

- **Polymorphism.** In essence, polymorphism is the ability access different objects through the same *interface*. For instance, if you have an interface that represents an electronic device, that interface would have the ability to turn the device on and off. You can use the actual physical types - computer, phone, television, etc - as if they were an electronic device, because they all have the on/off capability.

- **Inheritance.** This is one of the meatiest pieces of OO programming. Inheritance allows the sharing of BEHAVIORS. For instance, a Square is a type of Rectangle and has the same way to compute its area (width times height) - therefore by make Square inherit from Rectangle, we can share that behavior and avoid duplicating the code.

Example – Derived stocks



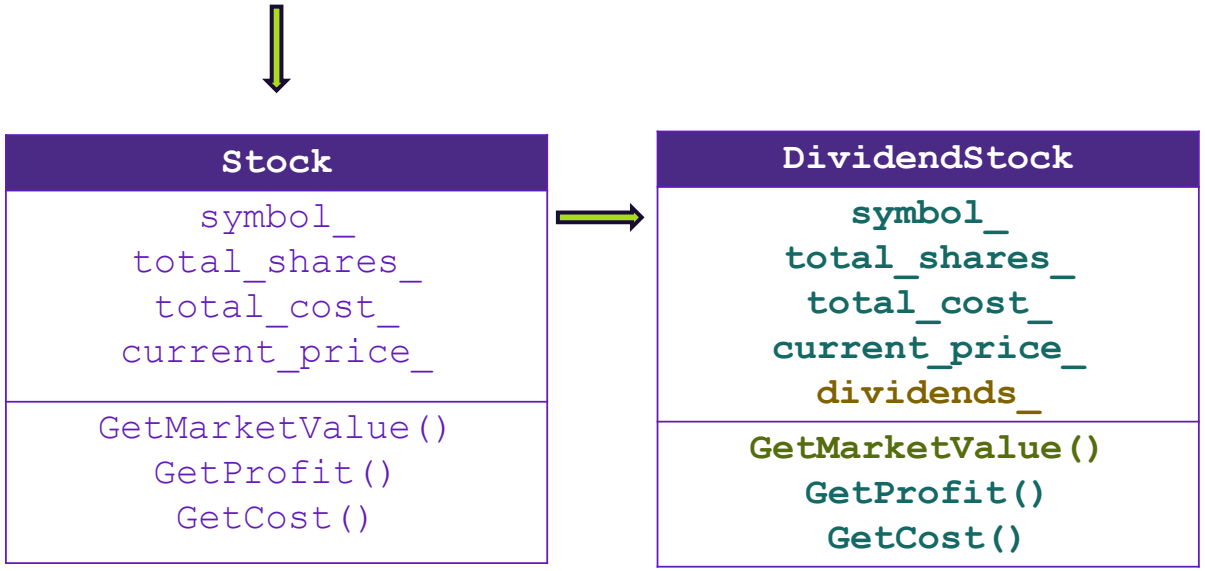
Here **Asset** is a **base** stock.

Cash **derives** from **Asset**

Stock **derives** from **Asset**

DividendStock **derives** from **Stock**

Example – Derived stocks details



A derived class can:

Inherit behaviors and state

Override some functions

Extend with new members

Dynamic Dispatch / Polymorphism

```
PromisedType* var_p = new ActualType();
```

- `var_p` is a **pointer** to an object of `ActualType` on the Heap
- `ActualType` must be the same or a derived class of `PromisedType`
- `PromisedType` defines the *interface* (*i.e.* what can be called on `var_p`), but `ActualType` may determine which *version* gets invoked

Analogy: A box labeled "cell phone" could hold Android or iPhone

- `PromisedType` is the box, `ActualType` is the Android or iPhone

Virtual or not?

In Java, all methods are `virtual`, why not in C?

- Efficiency:
  - Static function calls are a little faster
  - If there are no virtual functions you don't need a vptr
- Control:
  - You could *want* to call the function of the pointer type, not the created type

So, you can pick what you want, but be wary – it is unexpected to have a non-virtual method

Function Pointers

Code Globals Heap-> <-Stack

- Code also lives in memory
  - That means we can hold an address where a function's code resides
- **<return_type> (*<pointer_name>) (function_arguments);**
- **Set equal to 'address of function' (&f)**

```
double two(double x) {
  return 2.0;
}
double integrate(double (*f)(double), double lo,
  double hi, double delta) {
    //    ans += (*f)(x) * ((hi-lo) / (n+1));
    ans += f(x) * ((hi-lo) / (n+1));
    ….
int main() {
  integrate(&two, 0.0, 2.0, 1.0));
```

vtables & vptr

Question: How does this work?

Answer: We use function pointers!

If a class contains *any* virtual methods, the compiler emits:

- A (single) virtual function table (vtable) for *the class*
  - Contains a function pointer for each virtual method in the class
  - The pointers in the vtable point to the **most-derived** function for that class
- A virtual table pointer (vptr) for *each object instance*
  - A pointer to a virtual table as a "hidden" member variable
  - When the object's constructor is invoked, the vptr is initialized to point to the vtable for the newly constructed object's class
  - Thus, the vptr "remembers" what class the object is

Dynamic Dispatch vtables & vptrs

Stock vtable ptrs	function
getProfit	Stock::getProfit
getMarketValue	Stock::getMarketValue

DividendStock vtable ptrs	function
getProfit	Stock::getProfit
getMarketValue	DividentStock::getMarketValue

Object Vptrs	vtable
S	DividendStock vtable
Ds	DividendStock vtable

```
DividendStock dividend();
DividendStock* ds = &dividend;
Stock* s = &dividend;
// Calls DividendStock::GetMarketValue()
ds→GetMarketValue();
// Calls DividendStock::GetMarketValue()
s→GetMarketValue();

// Calls Stock::GetProfit().
// since that method is inherited
// Stock::GetProfit() calls
// DividendStock::GetMarketValue().
s→GetProfit();
// Calls Stock::GetProfit(),
// since that method is inherited.
// Stock::GetProfit() calls
// Dividendstock::GetMarketValue()
ds→GetProfit();.
```

Virtual? Constructing and Destructing

- Constructor of base class gets called *before* constructor of derived class
  - Default (zero-arg) constructor unless you specify a different one using initializer syntax
  - Initializer syntax: Foo::Foo(…): Bar(args); it(x) { … }
  - Needed to execute base class constructor with arguments; also works on instance variables and is preferred in production code (slogan: "initialization preferred over assignment")
- Destructor of base class gets called after destructor of derived class
  - It makes sense to make destructors `virtual` if you intend to derive from class
- So constructors/destructors really extend rather than override
  - Typically what you want
  - Java is the same

(Up) casting classes

- An object of a derived class *cannot* be cast to an object of a base class.
  - For the same reason a `struct T1 {int x,y,z;}` cannot be cast to type `struct T2 {int x,y;}` (different size)
- A pointer to an object of a derived class *can* be cast to a pointer to an object of a base class.
  - For the same reason a `struct T1*` can be cast to type `struct T2*` (pointers to a location in memory)
  - (Story not so simple with multiple inheritance)
- After such an *upcast*, field-access works fine (prefix)
  - but what do method calls mean in the presence of overriding? (see virtual)

(Down) casting classes

- C pointer-casts: unchecked; be careful
- Java: checked; may raise ClassCastException
- New: C++ has "all the above" (several different kinds of casts)
  - If you use single-inheritance and know what you are doing, the C-style casts (same pointer, assume more about what is pointed to) should work fine for downcasts
  - Worth learning about the differences on your own

Pure virtual methods & Abstract Classes

- Sometimes we want to include a function in a class but *only* implement it in derived classes
  - In Java, we would use an abstract method
  - In C++, we use a "**pure virtual**" function
    - Example: `virtual string noise() = 0;`
- A class containing *any* pure virtual methods is abstract
  - You can't create instances of an **abstract** class
  - Extend abstract classes and override methods to use them
- A class containing *only* pure virtual methods is the same as a Java interface
  - Pure type specification without implementations (e.g. `asset`)

Let's Practice

(which assignments work?)

```
baseclass* x = new
  baseclass();

derivedclass* y = new
  derivedclass();

baseclass* z = new
  derivedclass();

derivedclass* zz = new
  baseclass();
```

```
class baseclass{
public:
  baseclass() { cout << "construct baseclass()" << endl;}
  ~baseclass() { cout << "destruct ~baseclass" << endl; }
  void m1() { cout << "baseclass:m1" << endl; }
  void m2() { cout << "baseclass:m2" << endl; }
};

// class inherits from baseclass
class derivedclass : public baseclass{
public:
  derivedclass() { cout << "construct derived" << endl; }
  ~derivedclass() { cout << "destruct ~derived" << endl; }
  void m1() { cout << "derivedclass:m1" << endl; }
  void m2() { cout << "derivedclass:m2" << endl; }
  void m3() { cout << "derivedclass:m3" << endl; }
};
```

Let's Practice

(what functions are called?)

```
x→m1();
x→m2();
y→m2();
y→m3();
z→m2();
z→m3();
```

```
class baseclass{
public:
  baseclass() { cout << "construct baseclass()" << endl;}
  ~baseclass() { cout << "destruct ~baseclass" << endl; }
  void m1() { cout << "baseclass:m1" << endl; }
  void m2() { cout << "baseclass:m2" << endl; }
};
// class inherits from baseclass
class derivedclass : public baseclass{
public:
  derivedclass() { cout << "construct derived" << endl; }
  ~derivedclass() { cout << "destruct ~derived" << endl; }
  void m1() { cout << "derivedclass:m1" << endl; }
  void m2() { cout << "derivedclass:m2" << endl; }
  void m3() { cout << "derivedclass:m3" << endl; }
};
 baseclass* x = new baseclass();
  derivedclass* y = new derivedclass();
  baseclass* z = new derivedclass();
```

Let's Practice

(what functions are called? answers)

```
x→m1(); (base)
x→m2(); (base)
y→m2(); (der.)
y→m3(); (der.)
z→m2(); (base)
z→m3(); (err!)
```

```
class baseclass{
public:
  baseclass() { cout << "construct baseclass()" << endl;}
  ~baseclass() { cout << "destruct ~baseclass" << endl; }
  void m1() { cout << "baseclass:m1" << endl; }
  void m2() { cout << "baseclass:m2" << endl; }
};
// class inherits from baseclass
class derivedclass : public baseclass{
public:
  derivedclass() { cout << "construct derived" << endl; }
  ~derivedclass() { cout << "destruct ~derived" << endl; }
  void m1() { cout << "derivedclass:m1" << endl; }
  void m2() { cout << "derivedclass:m2" << endl; }
  void m3() { cout << "derivedclass:m3" << endl; }
};
 baseclass* x = new baseclass();
  derivedclass* y = new derivedclass();
  baseclass* z = new derivedclass();
```

Let's Practice

(what **virtual** functions are called?)

```
x→m1();
x→m2();
y→m1();
y→m2();
z→m1();
z→m2();
```

```
class baseclass{
public:
  baseclass() { cout << "construct baseclass()" << endl;}
  virtual ~baseclass() { cout << "~baseclass" << endl; }
  virtual void m1() { cout << "baseclass:m1" << endl; }
  void m2() { cout << "baseclass:m2" << endl; }
};
// class inherits from baseclass
class derivedclass : public baseclass{
public:
  derivedclass() { cout << "construct derived" << endl; }
  ~derivedclass() { cout << "destruct ~derived" << endl; }
  void m1() override { cout << "derivedclass:m1" << endl; }
  void m2() { cout << "derivedclass:m2" << endl; }
  void m3() { cout << "derivedclass:m3" << endl; }
};
 baseclass* x = new baseclass();
  derivedclass* y = new derivedclass();
  baseclass* z = new derivedclass();
```

Let's Practice

(what **virtual** functions are called? Answers)

```
x→m1(); (base)
x→m2(); (base)
y→m1(); (der.)
y→m2(); (der.)
z→m1(); (der.)
z→m2(); (base)
```

```
class baseclass{
public:
  baseclass() { cout << "construct baseclass()" << endl;}
  virtual ~baseclass() { cout << "~baseclass" << endl; }
  virtual void m1() { cout << "baseclass:m1" << endl; }
  void m2() { cout << "baseclass:m2" << endl; }
};
// class inherits from baseclass
class derivedclass : public baseclass{
public:
  derivedclass() { cout << "construct derived" << endl; }
  ~derivedclass() { cout << "destruct ~derived" << endl; }
  void m1() override { cout << "derivedclass:m1" << endl; }
  void m2() { cout << "derivedclass:m2" << endl; }
  void m3() { cout << "derivedclass:m3" << endl; }
};
 baseclass* x = new baseclass();
  derivedclass* y = new derivedclass();
  baseclass* z = new derivedclass();
```

NEXT UP

Templates

Overloaded functions

- Imagine that you want to write similar functions for different data types
- Function over-loading lets us define the same function with different types (`int`, or `string`)
- We re-use a lot of code!

```
// 0 if equal, 1 if value1 is bigger, else -1
int compare(const int& val1, const int& val2) {
  if (val1 < val2) return -1;
  if (val2 < val1) return  1;
  return 0;
}

// 0 if equal, 1 if value1 is bigger, else -1
int compare(const string& val1, const string& val2) {
  if (val1 < val2) return -1;
  if (val2 < val1) return  1;
  return 0;
}
```

Generic functions?

- These two implementations are nearly identical!
- What if we wanted `compare` for even more types? Or every comparable type?
- Prefer to write code that
  - Is *type-independent*
  - Is *Compile-type polymorphic*

```
// 0 if equal, 1 if value1 is bigger, else -1
int compare(const int& val1, const int& val2) {
  if (val1 < val2) return -1;
  if (val2 < val1) return  1;
  return 0;
}

// 0 if equal, 1 if value1 is bigger, else -1
int compare(const string& val1, const string& val2) {
  if (val1 < val2) return -1;
  if (val2 < val1) return  1;
  return 0;
}
```

C++ Parametric Polymorphism

C++ has the notion of **templates** (often referred to as *generics* elsewhere)

- A function or class that accepts a **type** as a parameter
  - You define the function or class once in a type-agnostic way
  - When you invoke the function or instantiate the class, you specify (one or more) types or values as arguments to it
- At **compile-time**, the compiler will generate the "specialized" code from your template using the types you provided
  - Your template definition is NOT runnable code
  - Code is *only* generated if you use your template

Function Templates

- We write a function that takes a datatype **T** as an argument
- Will work for any **T** for which **<** is defined.

```
#include <iostream>
#include <string>

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename T>   // < ... > can also be  <class T>
int compare(const T& value1, const T& value2) {
  if (value1 < value2) return -1;
  if (value2 < value1) return  1;
  return 0;
}
int main(int argc, char **argv) {
  std::string h("hello"), w("world");
  std::cout << compare<int>(10, 20) << std::endl;
  std::cout << compare<std::string>(h, w) << std::endl;
  std::cout << compare<double>(50.5, 50.6) << std::endl;
  return EXIT_SUCCESS;
}
```

Function Templates – type inference

- Compiler can also infer types…. usually

```
FunctionTemplate.cc:8:14:
  warning: comparison between
  two arrays [-Warray-compare]
     8 |   if (value2 < value1)
  return  1;

FunctionTemplate.cc:8:14: note:
  use unary '+' which decays
  operands to pointers or
  '&(value2)[0] < &(value1)[0]'
  to compare the addresses
```

```
#include <iostream>
#include <string>

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename T>   // <...> can also be  <class T>
int compare(const T& value1, const T& value2) {
  if (value1 < value2) return -1;
  if (value2 < value1) return  1;
  return 0;
}
int main(int argc, char **argv) {
  std::string h("hello"), w("world");
  std::cout << compare(10, 20) << std::endl; // infers int
  std::cout << compare(h, w) << std::endl; // infers string
  std::cout << compare("Hello", "World") << std::endl;
        // hm…
  return EXIT_SUCCESS;}
```

Function Templates with non-types

- You can also put non-types into a template.
  - Parameter is of fixed type
  - Use comma separated list to specify template arguments.

```
// Return a pointer to an N-element heap array
// (not entirely realistic…)
template <typename T, int N>
T* valarray(const T& val) {
  T* a = new T[N];
  for (int i = 0; i < N; ++i)
    a[i] = val;
  return a;
}
int main(int argc, char** argv) {
  int* ip = valarray<int, 10>(17);
  std::string* sp = valarray<std::string, 9>("hi");
  delete[] ip;
  delete[] sp;
  return EXIT_SUCCESS; }
```

How does it work?

The compiler doesn't generate any code when it sees the template function

- It doesn't know what code to generate yet, since it doesn't know what types are involved

When the compiler sees the function being used, then it understands what types are involved

- It generates the **instantiation** of the template and compiles it (kind of like macro expansion)
  - The compiler generates template instantiations for *each* type used as a template parameter

Class Templates

```
#ifndef PAIR_H_
#define PAIR_H_
#include <iostream>
#include <cstdlib>
template <typename Thing> class Pair {
 public:
  Pair() = default;  // initializes data members
  Thing get_first() const { return first_; }
  Thing get_second() const { return second_; }
  void set_first(const Thing& copyme);
  void set_second(const Thing& copyme);
  void Swap();
 private:
  Thing first_, second_;
};
#include "Pair.cc"
#endif  // PAIR_H_
```

- You can also template classes
  - Notice template before class
- This class holds a pair of any type of objects or primitives
- **`#include "Pair.cc"`** at the bottom – this is weird for a header file
  - Allows code to compile the Pair instance.
  - Could, instead, define Pair in header.
  - **`g++ -o pair usePair.cc`**

Class Templates – define code

```
#ifndef PAIR_H_
#define PAIR_H_
#include <iostream>
#include <cstdlib>
template <typename Thing> class Pair {
 public:
  Pair() = default;  // initializes members
  Thing get_first() const { return first_; }
  Thing get_second() const { return second_; }
  void set_first(const Thing& copyme);
  void set_second(const Thing& copyme);
  void Swap();
 private:
  Thing first_, second_;
};
#include "Pair.cc"
#endif  // PAIR_H_
```

```
template <typename Thing>
void Pair<Thing>::set_first(const Thing& copyme) {
  first_ = copyme;
}

template <typename Thing>
void Pair<Thing>::Swap() {
  Thing tmp = first_;
  first_ = second_;
  second_ = tmp;
}
template <typename T>
std::ostream& operator<<(std::ostream& out, const
  Pair<T>& p) {
  return out << "Pair(" << p.get_first() << ", " <<
  p.get_second() << ")";
}
```

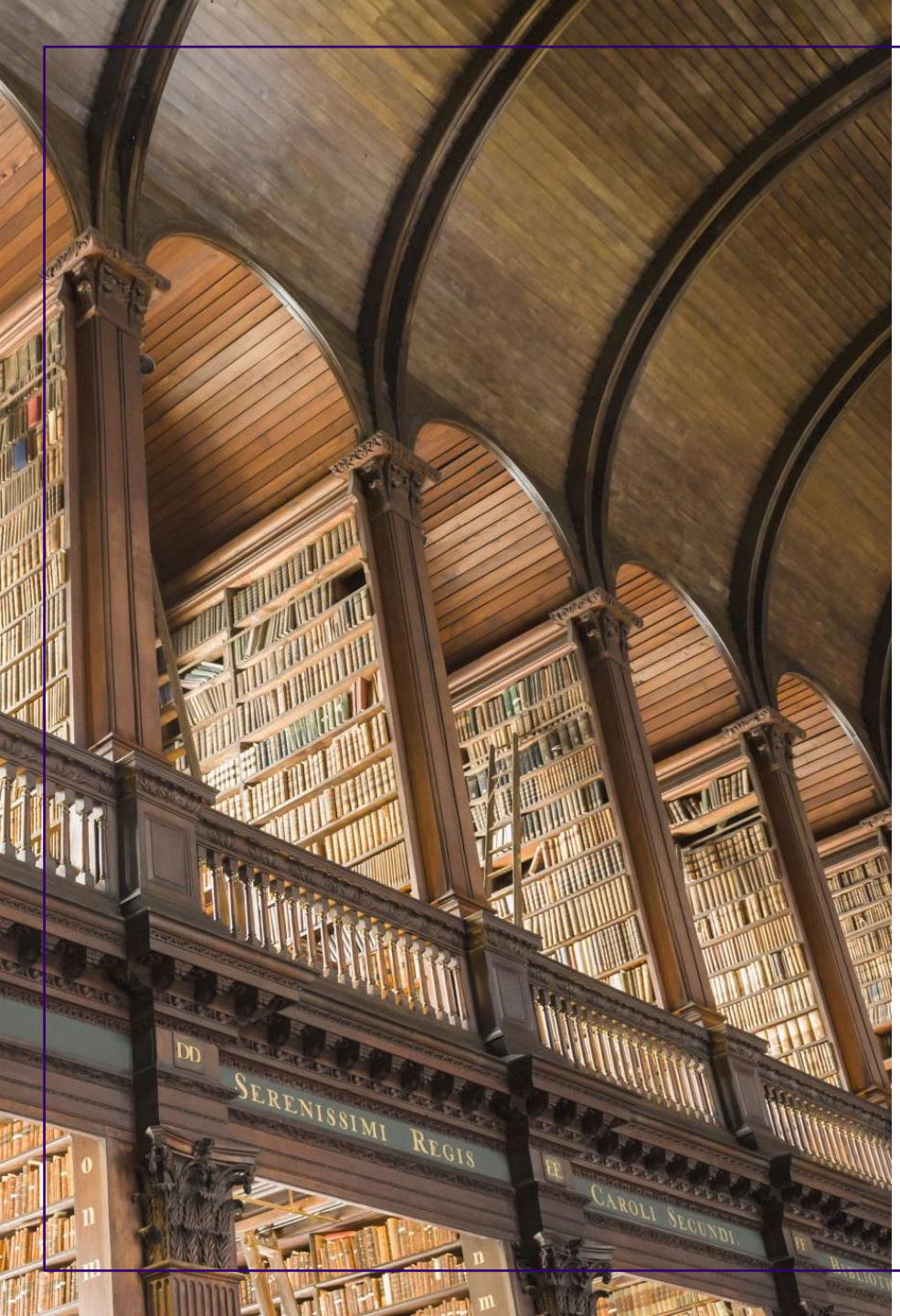
Class Templates – USING

```
#ifndef PAIR_H_
#define PAIR_H_
#include <iostream>
#include <cstdlib>
template <typename Thing> class Pair {
 public:
  Pair() = default;  // initializes members
  Thing get_first() const { return first_; }
  Thing get_second() const { return second_; }
  void set_first(const Thing& copyme);
  void set_second(const Thing& copyme);
  void Swap();
 private:
  Thing first_, second_;
};
#include "Pair.cc"
#endif  // PAIR_H_
```

```
#include "Pair.h"
using std::cout;
using std::endl;
using std::string;
int main(int argc, char** argv) {
  Pair<string> ps;  // this usage instantiates a
  class Pair<std::string>
  string x("foo"), y("bar");

  ps.set_first(x); // first_ = "foo", second_ = ""
  ps.set_second(y);//  second_ = "bar"
  ps.Swap();       // first_= "bar", second_= "foo"
  cout << ps << endl;  // nonmember overloaded
        operator<< invoked

  return EXIT_SUCCESS;
}
```



STANDARD TEMPLATE LIBRARY (STL)

C++ Standard Library

C++'s Standard Library consists of four major pieces:

1. The entire C standard library
2. C++'s input/output stream library
   - std::cin, std::cout, stringstream, fstream, etc.
3. C++'s standard template library (**STL**) 👉
   - Containers, iterators, algorithms (sort, find, etc.), numerics
4. C++'s miscellaneous library
   - Strings, exceptions, memory allocation, localization

C++ stdlib is built around templates

Containers store data using various underlying data structures

- The specifics of the data structures define properties and operations for the container

Iterators allow you to traverse container data

- Iterators form the common interface to containers
- Different flavors based on underlying data structure

Algorithms perform common, useful operations on containers

- Use the common interface of iterators, but different algorithms require different 'complexities' of iterators

STL Containers

A container is an object that stores (in memory) a collection of other objects (elements)

- Implemented as class templates, so hugely flexible

Several different classes of container

- Sequence containers (`vector, deque, list`, …)
- Associative containers (`set, map, multiset, multimap, bitset`, …)
- Differ in algorithmic cost and supported operations

Common C++ STL Containers

Sequence containers can be accessed sequentially

- **`vector<Item>`** uses a dynamically-sized contiguous array (like `ArrayList`)
- **`list<Item>`** uses a doubly-linked list (like `LinkedList`)

Associative containers use search trees and are sorted by keys

- **`set<Key>`** only stores keys (like `TreeSet`)
- **`map<Key, Value>`** stores key-value pair<>'s (like `TreeMap`)

Unordered associative containers are hashed

- **`unordered_map<Key, Value>`** (like `HashMap`)

STL Containers how-to

STL containers store by **value**, not by reference

- When you insert an object, the container makes a **copy**
- If the container needs to rearrange objects, it makes copies
  - *e.g.* if you sort a vector, it will make many, many copies
  - *e.g.* if you insert into a map, that may trigger several copies
- What if you don't want this (disabled copy constructor or copying is expensive)?
  - You can insert a wrapper object with a pointer to the object
    - We'll learn about these "smart pointers" soon

DEMOS

- Tracer class (for demonstration purposes)
- STL vector
- STL iterator
- STL algorithms

Tracer Class

Wrapper class for an `int value_`

- Holds unique `int id_` (increases from `0`)
- Default ctor (set unique `id_` for each instance), cctor, dtor, `op=`, `op<` defined
- `friend` function `operator<<` defined
- Private helper method `PrintID()` to return `"(id_,value_)"` as a string
- Class and member definitions can be found in `Tracer.h` and `Tracer.cc`

Useful for tracing behaviors of containers

- All methods print identifying messages
- Unique `id_` allows you to follow individual instances

```
class Tracer {
 public:
  Tracer();
  ~Tracer();
  Tracer(const Tracer &rhs
  // Assignment operator.
  Tracer &operator=(const Tracer &rhs);
  // Less-than comparison operator.
  bool operator<(const Tracer &rhs) const;
  // << operator
  friend std::ostream &operator<<(std::ostream &out, const
 Tracer &rhs);
 private:
  // Return "id_ [history_]" as a string
  std::string Print(void) const;
  // This static (class member) tracks the next id_ to hand
 out.
  static int nextid_;
  const int id_;  // fixed id of this Tracer
  int value_;     // the most recent id
};
```

STL vector

- A generic, dynamically resizable array
- https://cplusplus.com/reference/vector/vector/
- Elements are stored in **contiguous** memory locations
  - Like a normal C array, or the ArrayList in Java!
  - Elements can be accessed using pointer arithmetic if you'd like
  - Random access is O(1) time
    - Pointer arithmetic, then access
- Adding/removing from the end is cheap (amortized constant time)
- Inserting/deleting from the middle or start is expensive (linear time)
  - Need to shift all of the elements in the array

```
#include <vector>
int main(int argc, char **argv) {
  Tracer a, b, c;
  vector<Tracer> vec;

  cout << "vec.push_back " << a << endl;
  vec.push_back(a);
  cout << "vec.push_back " << b << endl;
  vec.push_back(b);
  cout << "vec.push_back " << c << endl;
  vec.push_back(c);

  cout << "vec[0]" << endl;
  cout << vec[0] << endl;

  cout << "vec[2]" << endl;
  cout << vec[2] << endl;
  return EXIT_SUCCESS;
}
```

STL iterator

Each container class has an associated iterator class (*e.g.* `vector<int>::iterator`) used to iterate through elements of the container

https://cplusplus.com/reference/iterator/

- Iterator range is from `begin` up to `end` i.e., [`begin`, `end`)
  - `end` is one past the last container element!
- Some container iterators support more operations than others
  - All can be incremented (`++`), copied, copy-constructed
  - Some can be dereferenced on RHS (*e.g.* `x = *it;`)
  - Some can be dereferenced on LHS (*e.g.* `*it = x;`)
  - Some can be decremented (`--`)
  - Some support random access (`[]`, `+`, `-`, `+=`, `-=`, `<`, `>` operators)

```
#include <vector>

int main(int argc, char **argv) {
  Tracer a, b, c;
  vector<Tracer> vec;
  vec.push_back(a);
  vec.push_back(b);
  vec.push_back(c);

  cout << "Iterating:" << endl;
  vector<Tracer>::iterator it;
  for (it = vec.begin(); it < vec.end();
       it++) {
    cout << *it << endl;
  }
  cout << "Done iterating!" << endl;
  return EXIT_SUCCESS;
}
```

Type inference & Range for

The auto keyword can be used to infer types

- Simplifies your life if, for example, functions return complicated types
- The expression using auto must contain explicit initialization for it to work

```
// Inferred type
auto facts2 = Factors(12321);

// Compiler error here
auto facts3;
```

Syntactic sugar similar to Java's `foreach`

- declaration defines loop variable
- expression is an object representing a sequence
  - Strings, initializer lists, arrays with an explicit length defined, STL containers that support iterators

```
for ( declaration : expression ) {
  statements
}
// Prints out a string, one
// character per line
std::string str("hello");
for ( auto c : str ) {
  std::cout << c << std::endl;
}
```

STL *updated* iterator

Using newer language features such as ‘auto’ and the ‘range-for’ makes for cleaner code!

```
#include <vector>

int main(int argc, char **argv) {
  Tracer a, b, c;
  vector<Tracer> vec;
  vec.push_back(a);
  vec.push_back(b);
  vec.push_back(c);

  cout << "Iterating:" << endl;
  // "auto" is a C++11 feature not available
  on older compilers
  for (auto &p : vec) {
    cout << p << endl;
  }
  cout << "Done iterating!" << endl;
  return EXIT_SUCCESS;
}
```

STL algorithms

A set of functions to be used on ranges of elements

- Range: any sequence that can be accessed through *iterators* or *pointers*, like arrays or some of the containers
- General form: `algorithm(begin, end, ...);`

Algorithms operate directly on range *elements* rather than the containers they live in

- Make use of elements' copy ctor, =, ==, !=, <
- Some do not modify elements
  - *e.g.* `find, count, for_each, min_element, binary_search`
- Some do modify elements
  - *e.g.* `sort, transform, copy, swap`

```
#include <vector>
#include <algorithm>
#include <cstdlib>
#include "Tracer.h"
using namespace std;

void PrintOut(const Tracer &p) {
  cout << "  printout: " << p << endl;
}
int main(int argc, char **argv) {
  Tracer a, b, c;
  vector<Tracer> vec;
  vec.push_back(c);
  vec.push_back(a);
  vec.push_back(b);
  sort(vec.begin(), vec.end());
  cout << "done sort!" << endl;
  for_each(vec.begin(), vec.end(), &PrintOut);
  return EXIT_SUCCESS;  }
```

Others

- `<list>` A generic doubly-linked list
- https://cplusplus.com/reference/list/list/
- Elements are **not** stored in contiguous memory locations
  - Does not support random access (*e.g.* cannot do list[5])
- Some operations are much more efficient than vectors
  - Constant time insertion, deletion anywhere in list
  - Can iterate forward or backwards
    - Backward: --
    - Forward: ++
- Has a built-in sort member function
  - Doesn't copy! Manipulates list structure instead of element values

- `<map>` One of C++'s *associative* containers: a key/value table, implemented as a search tree
- https://cplusplus.com/reference/map/
- General form: `map<key_type, value_type> name;`
- Keys must be *unique*
  - multimap allows duplicate keys
- Efficient lookup (O(log n)) and insertion (O(log n))
  - Access value via name[key]
    - If key doesn't exist in map, it is added to the map
- Elements are type pair<key_type, value_type> and are stored in *sorted* order (key is field first, value is field second)
- Key type must support less-than operator (<)