C++ Models

Jim Fawcett

https://JimFawcett.github.io

Background

- The material for this presentation comes from my github website:
  - https://JimFawcett.github.io
- The site provides a curated selection of code developed for graduate software design courses at Syracuse University
- It also contains tutorial and reference materials related to that code.
- Some of that is presented in the form of "stories"
- C++ Models is the title of the first chapter of a "C++ Story"
  - The story is a detailed walk-through the C++ programming language. It provides reference material for a set of repositories – 61 at last count – that hold source code for utilities, tools, components, and demonstrations.

Prologue

- C++ is a large and ambitious language.
- Models help us understand important features of the language
  - Show the language's internal consistency
  - Help us understand and use it effectively
- We will consider:
  - Code Structure, Compilation, and Execution
  - Use of memory
  - Classes and the C++ object model
  - Templates

Presentation Resources

- The ideas discussed in this presentation are drawn from a web page:
  https://JimFawcett.github.io/CppStory_Models.html

  which is part of the CppStory:
  https://JimFawcett.github.io/CppStory_Prologue.html

- And code examples for the story are documented here:
  https://JimFawcett.github.io/CppStoryRepo.html

- These slides are available here:
  https://JimFawcett.github.io/Resources/CppModels.pdf

1. Code Structure

https://jimfawcett.github.io/CppStory_Models.html#structure

- Source code is written in files
- For many software systems file structures become large and hard to understand.
- To support readability and maintenance, we create packages that consist of a few files with a single purpose and document the purpose and design in comments.
  - Files are units of construction
  - Packages are units of documentation
  - The C++ build system does not recognize packages, but it does recognize projects.
  - Packages are simply groups of one or more files, stored in a single directory, annotated with comments to support understanding, test, and maintenance, with a project for building.
- Packages are units of documentation and translation
  - Each has a Visual Studio project or make file

Packages

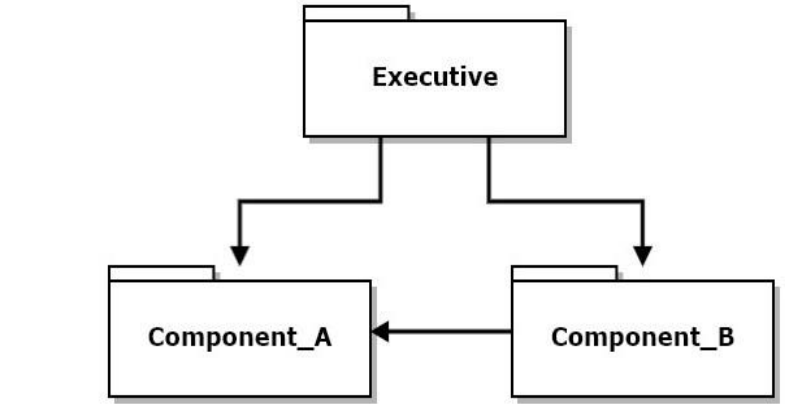
- A package consists of:
  - a single Package_Name.cpp file with construction test main function
  - a header file Package_Name.h
  - Optionally may have an interface file, IPackage_Name.h
- These parts are embedded in a directory with project file or make file
  - Used to build the construction test.
  - The directory may also include files from other packages on which this package depends.
  - Alternately, a project package makes references to libraries for packages on which it depends instead of including the files in its package directory.
- Each package is expected to implement a single responsibility and have code comments that describe its operation.

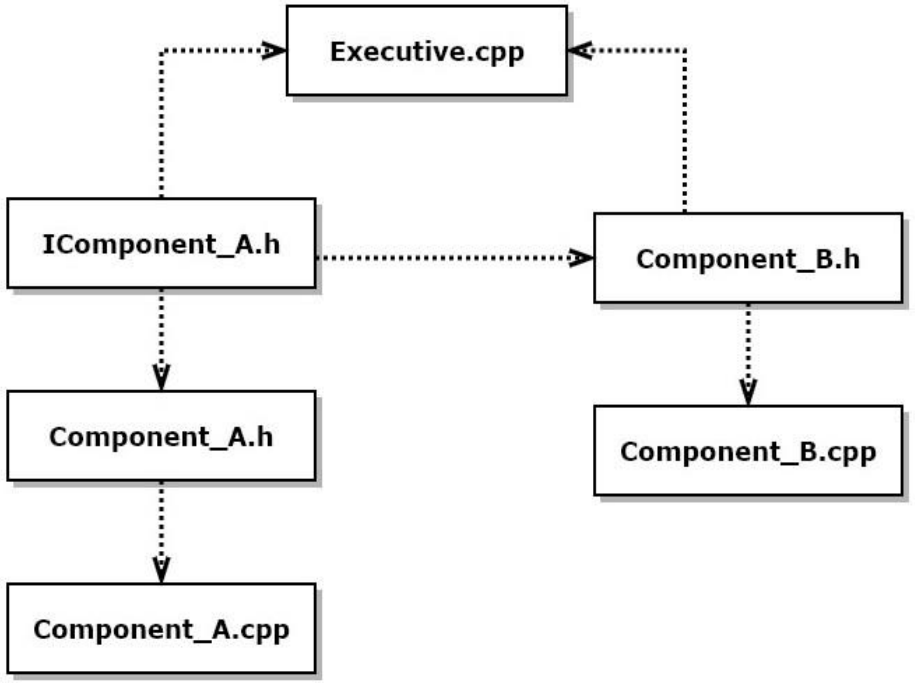
Package Construction Co-Tests

- For anything other than trivial example code it's very useful to test as we build code:
  - Add a main function for every .cpp file.
  - Every time we add a few lines of code we add small tests in the main then build and execute.
  - This "co-test" process allows us to very quickly find errors. If a test fails, the problem is almost certain to be in the few lines of code we entered after the last test.
  - We wrap the main function in #ifdef TEST_NAME … #endif directives.
    - When TEST_NAME is defined we run the package test.
    - When not defined we can combine the package with other packages to build a larger executable.

Example – Files and Packages

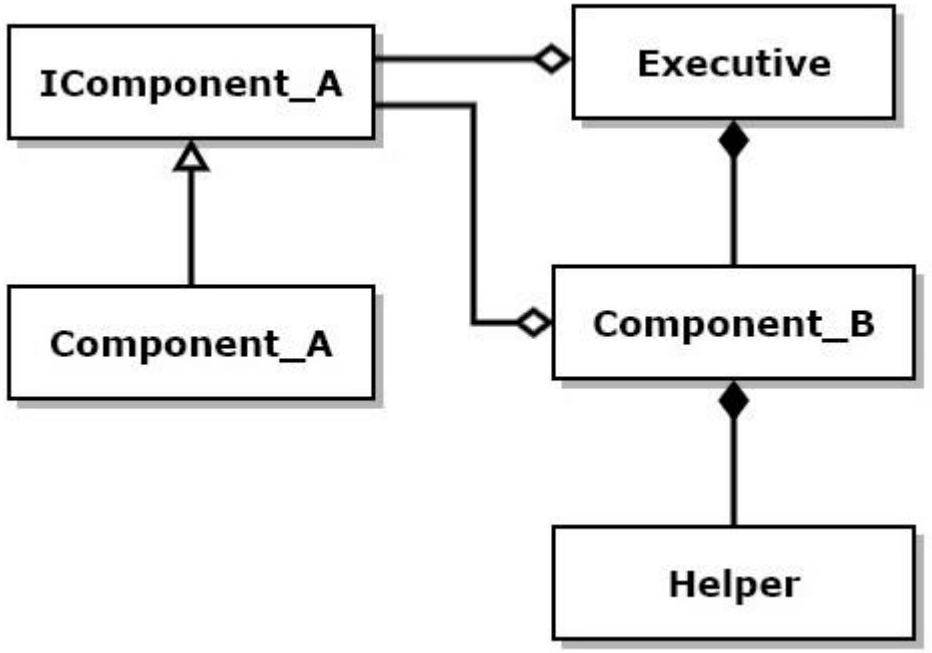
- The diagram at the top shows a set of packages that represent the files shown at the bottom.
- The file diagram shows file include relationships.
  - Both Executive.cpp and Component_B.cpp include a header file IComponent_A.h
  - Include files provide type information that is declared elsewhere.
- The Component_A package contains all the files with names containing Component_A.
- Code for this example: https://github.com/JimFawcett/CppStory in Chapter1-Structure





Example - Classes

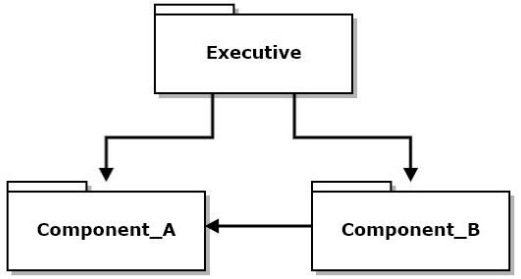
- This diagram shows classes that are defined in each of the files from the previous slide.
  - IComponent_A is an interface[1] for Component_A
  - Component_A implements the interface to provide exported services
  - Component_B doesn't provide an interface, composes class Helper
  - Component_B uses Component_A through its interface and factory[2]
  - Executive uses Component_A through its interface, composes Component_B



---

1. C++ does not have an interface construct. We use structs with pure virtual functions for that purpose.
2. Component_A's factory is implemented with a function, declared in IComponent_A.h. and implemented in Component_A.cpp

Use of Interfaces and Factories



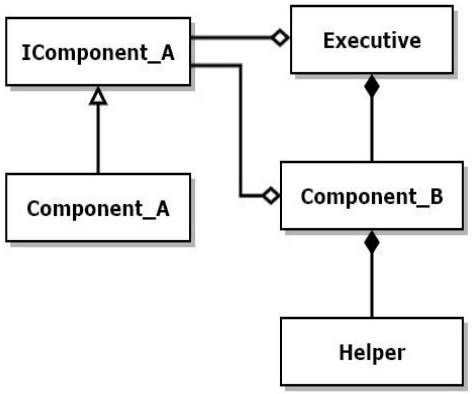
- If you look at interface IComponent_A.h you will see it has no implementation detail.

```
struct IComponent_A {
  virtual ~IComponent_A() {}
  virtual void say() = 0;
};
```

```
inline std::unique_ptr<IComponent_A>
createComponent_A(const std::string& id);
```

- That means that Executive and Component_B have no build dependencies on Component_A. Component_A can change any of its implementation without affecting Executive or Component_B as long as the interface, IComponent_A and factory function signature don't change.

Object Factories



- There are two kinds of object factories:
  1. Factories that return a smart pointer referring to a newly created instance of a component in the native heap.

     std::unique_ptr<IComponent> createComponent() { … }

  2. Singleton factories that return a reference to a static component.

     IComponent& getComponentInstance() { … }

- You can find examples of both in the Logger repository:

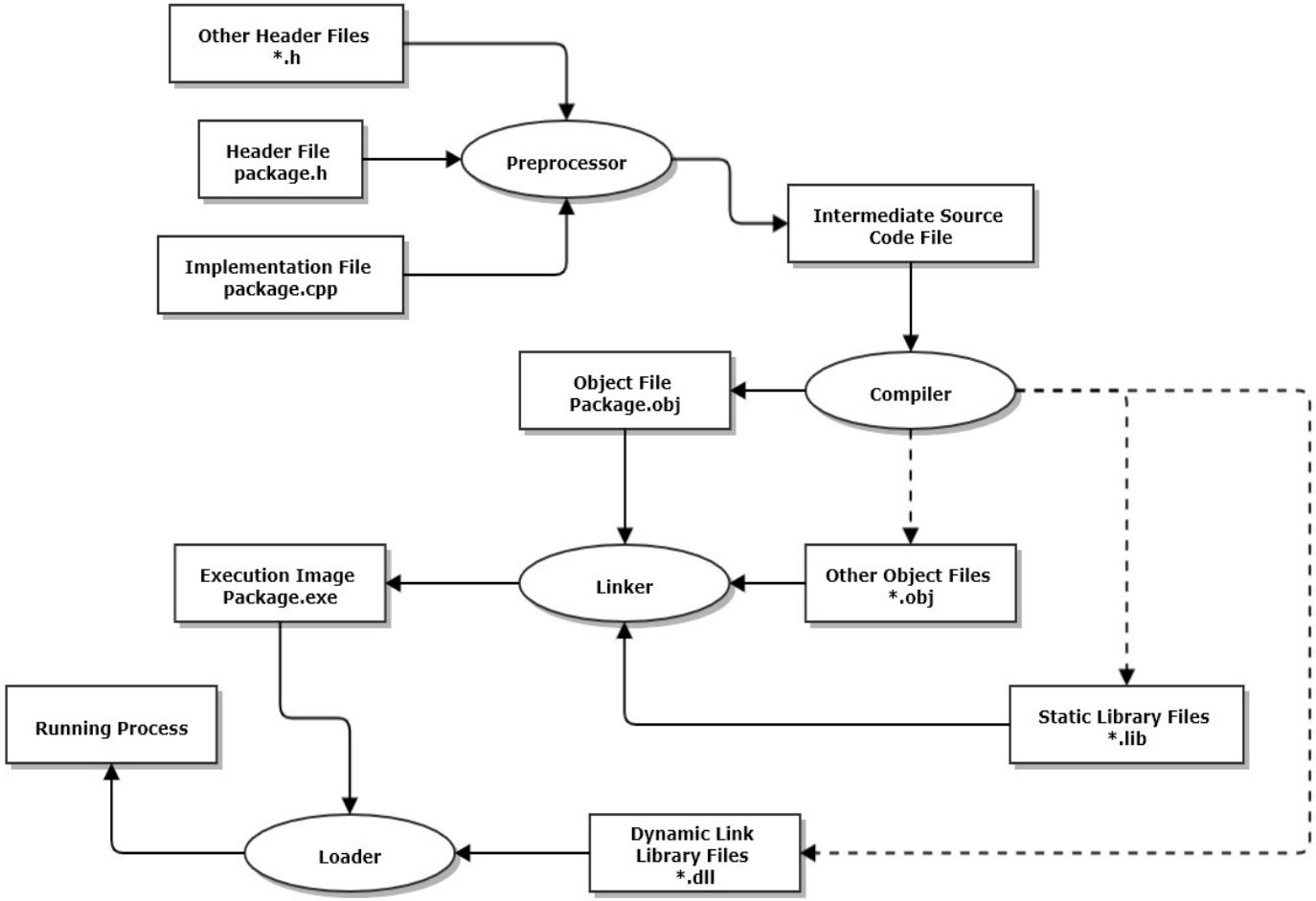
  https://JimFawcett.github.io/Logger.html

  in files ITestLogger.h and IQTestLogger.h

2. Compilation Model

https://jimfawcett.github.io/CppStory_Models.html#compil

- The C++ build process translates each *.cpp file independently.
- We say that a *.cpp file and all its include files are a translation unit.
- Translation begins by inserting the contents of each included *.h file into the *.cpp file at the site of the include.
- That is then compiled into an object file, *.obj.
- That process is repeated for all cpp files in the current build.
- The linker then binds the obj files into an executable.



Compilation Model

- One consequence of this build process is the definition first rule.
- The C++ language was designed to support one-pass compilation.
- The compiler can't layout code for an instance until it knows the instance's size.
- That comes from seeing the class, struct, or enum declaration.
- You can create a pointer to an incomplete type, e.g., forward declaration like `class A;` but you can't use it until after the type is completed with a declaration.

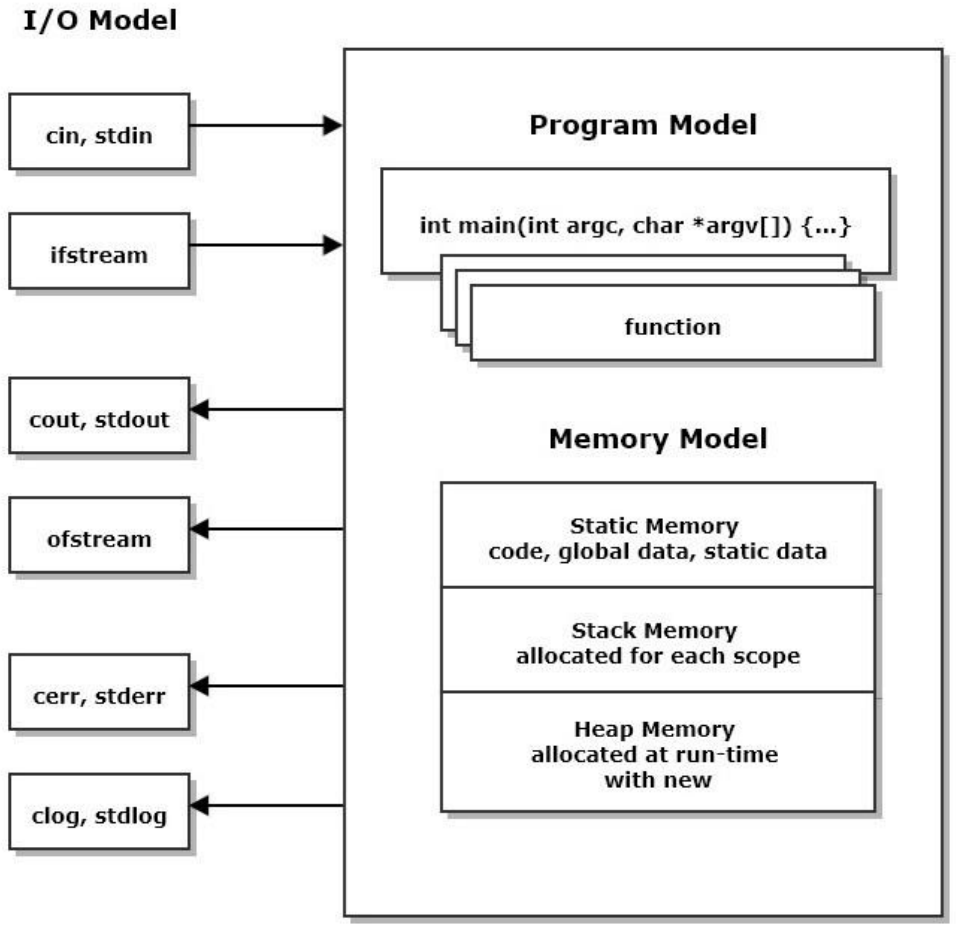
- Definition First Rule
  - Instances of classes, structs, and enums can be created only after those entities are declared.

- If you think of a cpp file as an ocean of syntax and its include files as syntax tributaries filling the ocean, then the entity declarations must be upstream from the point of entity creation. Here, upstream simply means compiler scan order.

3. Program Execution

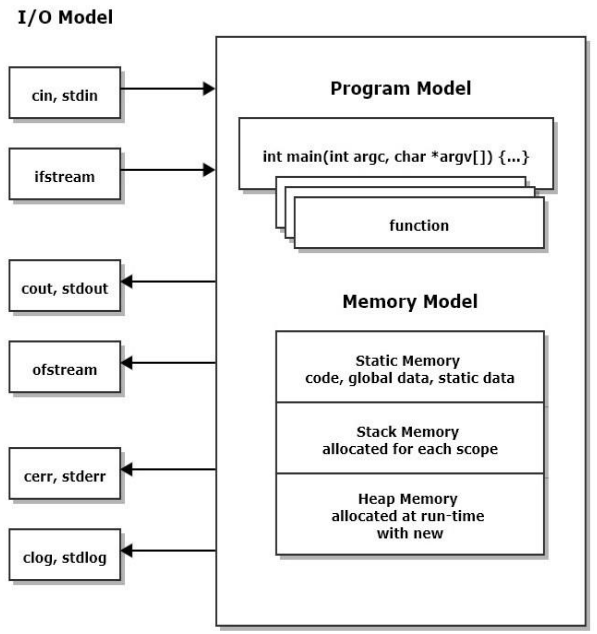
https://jimfawcett.github.io/CppStory_Models.html#execute

- When the executable for a program is loaded:
  - Initialization code provided by the compiler executes
  - Then the function main is entered.
    - main is just a function that is defined to the linker as the entry point for processing.
    - Some project types will use other names, e.g., WinMain
- Any function may call other functions within the executable.



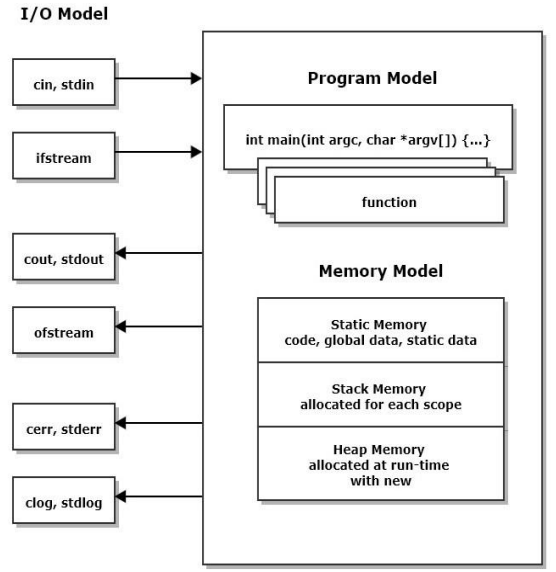
Use of program memory

- When the thread of execution enters a function an allocation of stack memory is used to store function parameters and any local data defined in the function.
  - The same thing happens for every scope, defined by a matching pair of braces, { and }. For example, an if statement, using braces, allocates stack memory to hold data local to its scope.
- A program may place any of its entities, e.g., an instance of a user-defined class, into static memory, stack memory, or heap memory.
- We will discuss the consequences of that in the next section.



Interaction with the Execution Environment

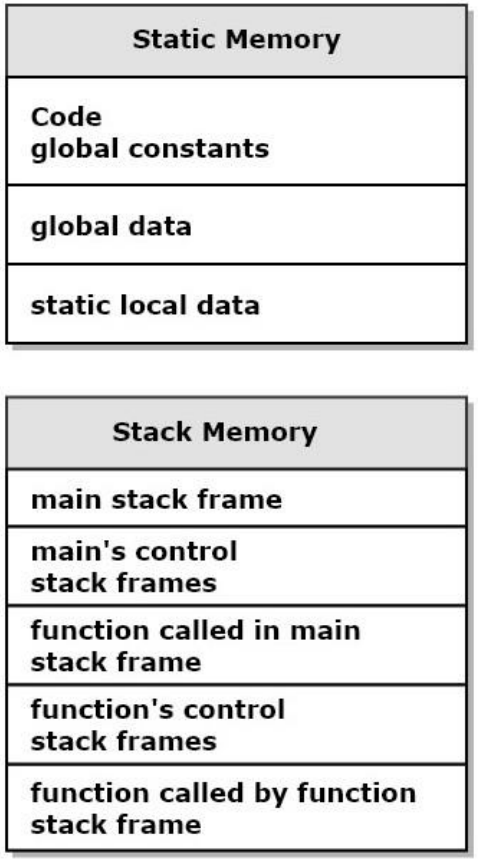
- There are two primary ways for a C++ program to observe and use its execution environment:
  - Use a stream object like std::cout or std::cin.
  - Classes for streams are provided by the standard library, via include statements:
    #include<iostream>, #include<fstream>, …
  - The std::iostream library defines global objects:
    std::cout, std::cin, std::cerr, std::clog
    that are always accessible to a program.
- The program may use services of its platform API by including certain header files specific to each platform, e.g., windows, linux, …



4. Memory Model

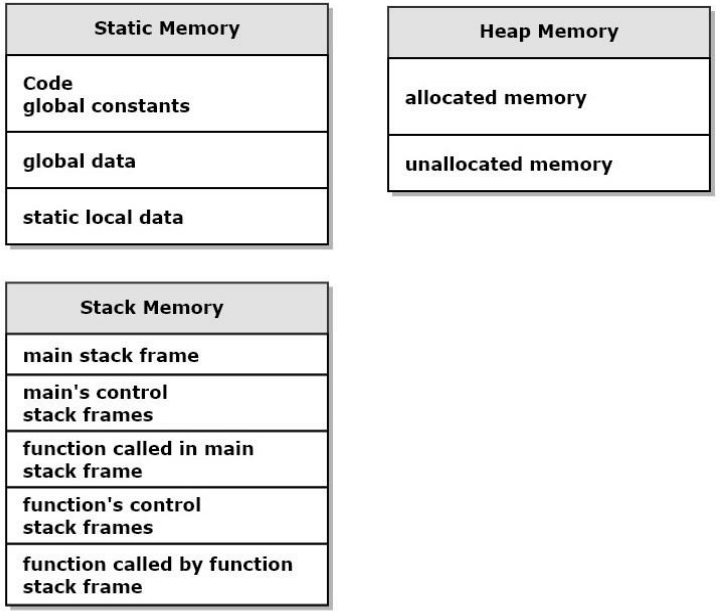
https://JimFawcett.github.io/CppStory_Models.html#memory

- Static memory is used to store code and entities that live for the entire program execution
- Stack memory is used as scratch-pad to store information needed in each scope, e.g., local data. It becomes invalid when the thread of execution leaves the scope.
- Heap memory is used to store entities that live from the time the program creates them with a call to new until the program discards them with a call to delete



Control of entity placement in memory

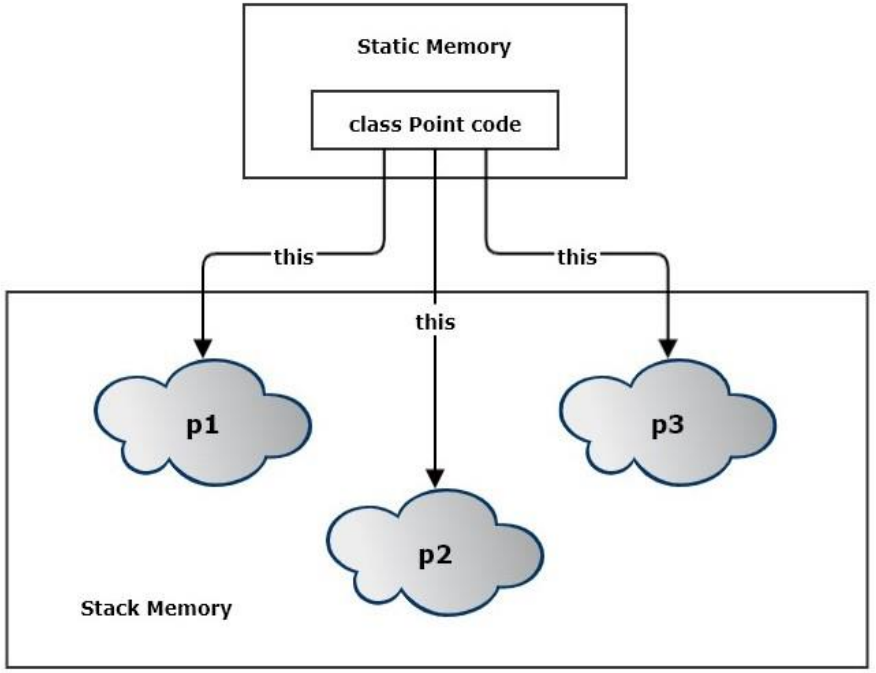
- The compiler places all code and global data in static memory.
- A program can place an entity instance in static memory by qualifying its declaration with the keyword static.
- C++ code also places entities in stack memory by calling a function, placing function parameters and local data in its stack frame.
- Also, every local scope, defined by braces, { and }, creates a new allocation of stack memory to hold data local to that scope.
- An entity instance is placed in heap memory by a call to new and removed with a call to delete.



5. Classes

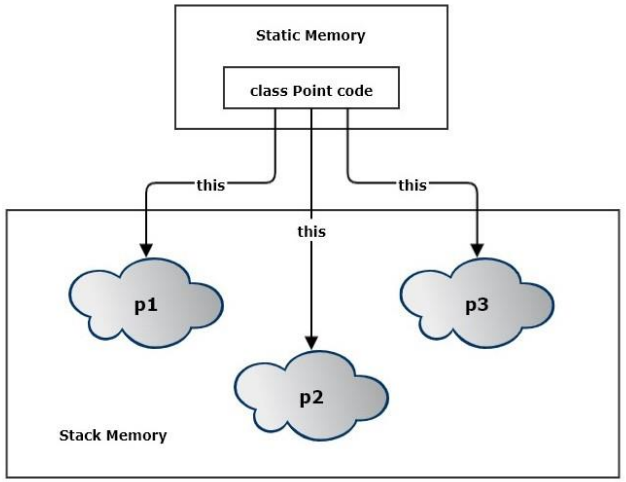
https://jimfawcett.github.io/CppStory_Models.html#class

- Classes are units of data management.
- A class is a “cookie cutter” for stamping out class instances in memory.
- Each instance has member data that has been initialized by a class constructor.
- When an instance method is invoked, say p1.name(“p1”), the code in static memory needs to know which instance invoked its name method. Each call of a non-static method sends its address to its code in static memory in an implicit argument called “this”. The code may then mutate p1’s state as defined by the method using this pointer.



Point Class

- A class designed to represent points in some space might be declared as shown on the right.
- The space might represent physical space-time, so the coordinates might be physical x, y, z, t values representing width, depth, height, and time.
- If we declare an instance of Point locally in some function, the member data of the class will be stored in static memory.
  - However, the std::vector<double> stores its contents, the coordinates in this case, in the native heap.
  - Strings do the same thing. All string characters are also stored in the heap.
  - The size of this object, used by the compiler to set up allocation for the function's stack frame, is just the static memory consumed by each data member. It does not include heap allocations, because that is allocated at run time, not by the compiler. This is what the sizeof operator measures.

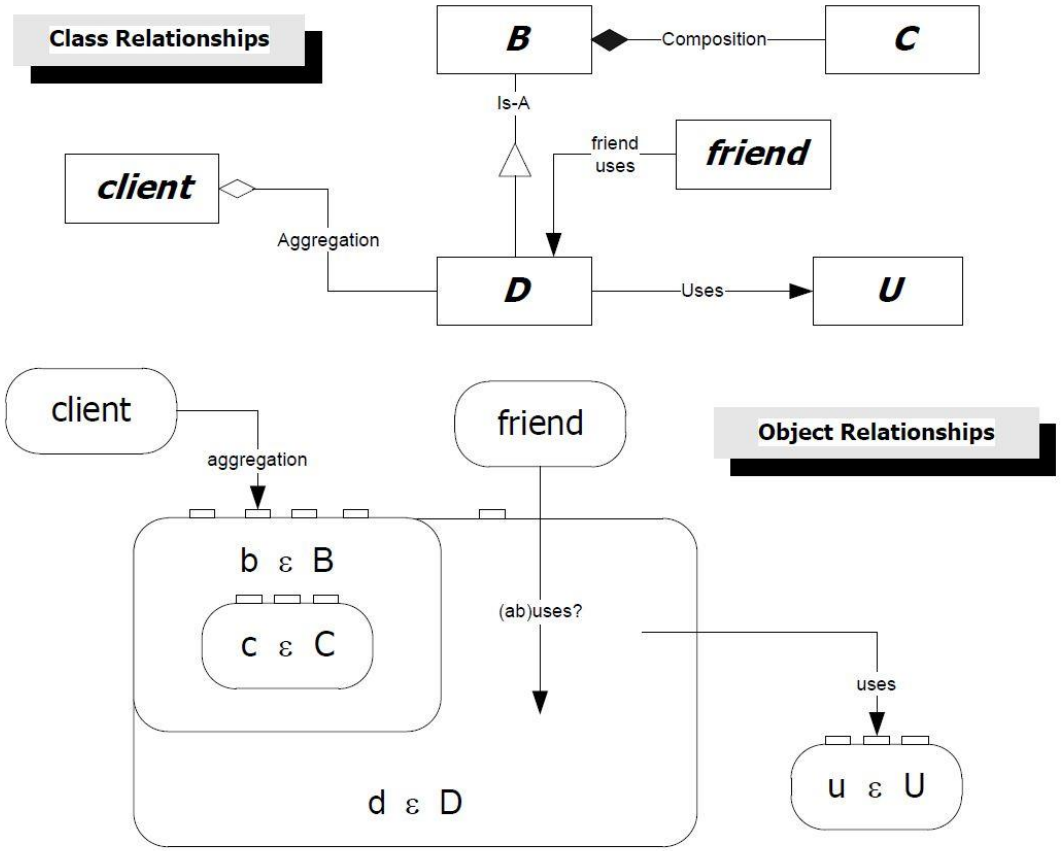



```
class Point {
public:
  using iterator = std::vector<double>::iterator;
  using const_iterator =
    std::vector<double>::const_iterator;
  Point(const std::string& name = "none");
  Point(std::initializer_list<double> il);
  void name(const std::string& name);
  std::string name() const;
  double& operator[](size_t i);
  double operator[](size_t i) const;
  size_t size() const;
  iterator begin();
  iterator end();
  const_iterator begin() const;
  const_iterator end() const;
private:
  std::string name_ = "unspecified";
  std::vector<double> coordinates_;
};
```

6. C++ Object Model

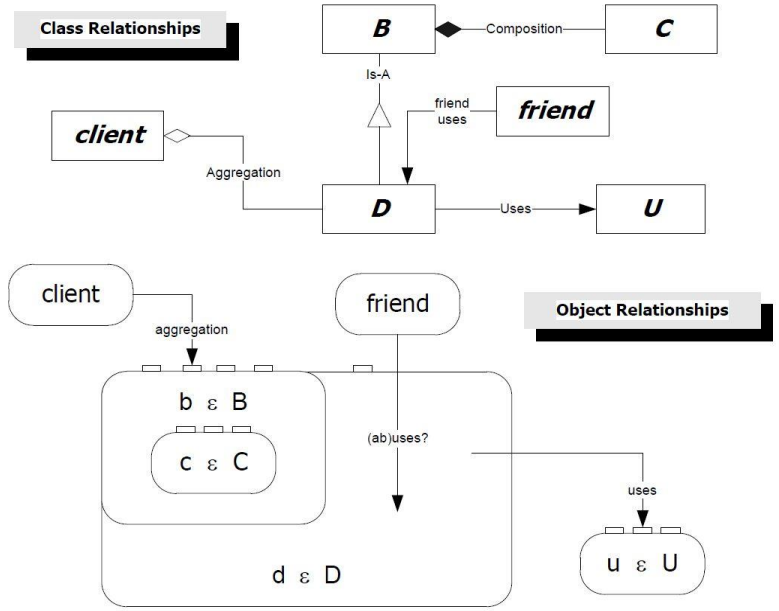
https://jimfawcett.github.io/CppStory_Models.html#objmodel

- The C++ object model is concerned with how compound objects are laid out in memory.
- Structs and classes support five relationships that bind objects together to build compound objects: inheritance, composition, aggregation, using, and friend-ship.
  - Inheritance causes base class instances to be encapsulated within the memory footprint of instances of classes that derive from them.
  - Composition does the same thing. A composing class instance contains an instance of each class it composes.
  - These are strong owning relationships. Weak ownership – aggregation, and non-owning relationships - using, and friend-ship - do not cause this encapsulation.



Object Construction

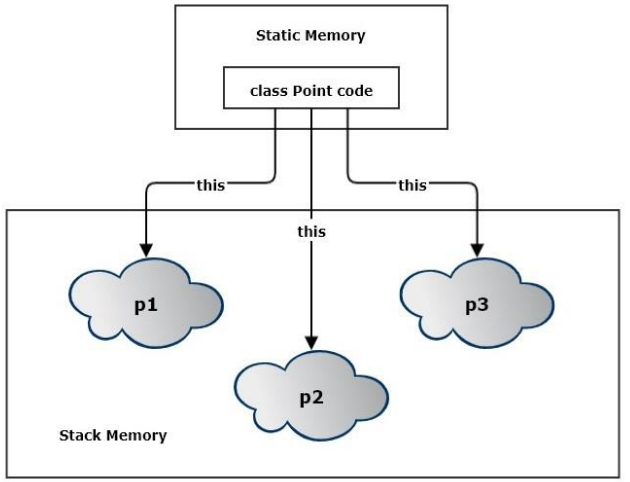
- When B is constructed its C component is constructed in its memory footprint. That means that C is constructed as part of B's construction.
- When D is constructed its base B is constructed in its memory footprint. So B is constructed as part of D's construction.
- These are required events that affect the syntax of the constructors we write.
  - We use an initialization sequence for B to determine how C is to be constructed.
  - Similarly, D uses an initialization sequence to determine how B is constructed.



- Code for this example: https://github.com/JimFawcett/CppStory in Chapter1-Classes
- Documented in: https://JimFawcett.github.io/CppStoryRepo.html

Value Types

- Value types have **instances** that can be copied and assigned.
- C++ has been designed from the beginning to support creation of user-define value types.
  - If class member data and base member data have correct copy and assignment semantics, copies are made by copying each of the class and base members, and for assignment the process is the same. Examples are classes with only fundamental types and STL containers as data members.
  - If class members do not have correct copy and assignment semantics C++ supports the definition of copy constructors and assignment operators that the developer designs to provide correct copy and assignment operations. Examples are classes that contain pointer data members.
- Point is a value type.

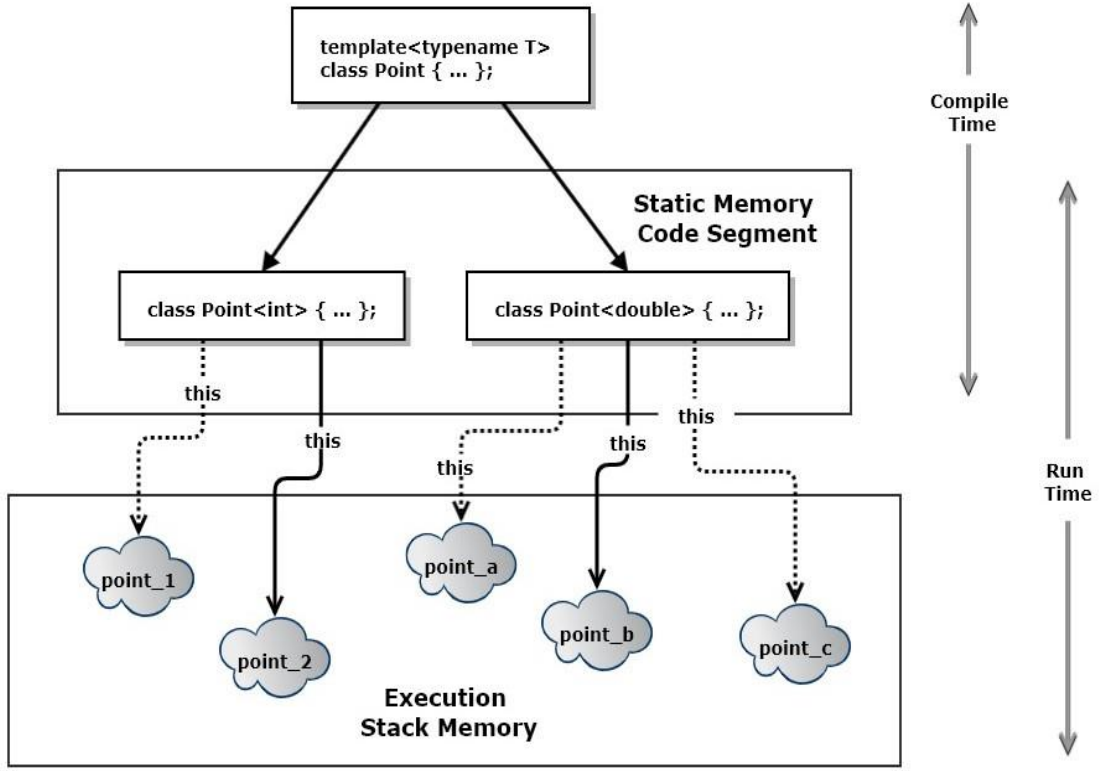



```
class Point {
public:
  using iterator = std::vector<double>::iterator;
  using const_iterator =
    std::vector<double>::const_iterator;
  Point(const std::string& name = "none");
  Point(std::initializer_list<double> il);
  void name(const std::string& name);
  std::string name() const;
  double& operator[](size_t i);
  double operator[](size_t i) const;
  size_t size() const;
  iterator begin();
  iterator end();
  const_iterator begin() const;
  const_iterator end() const;
private:
  std::string name_ = "unspecified";
  std::vector<double> coordinates_;
};
```

7. Templates

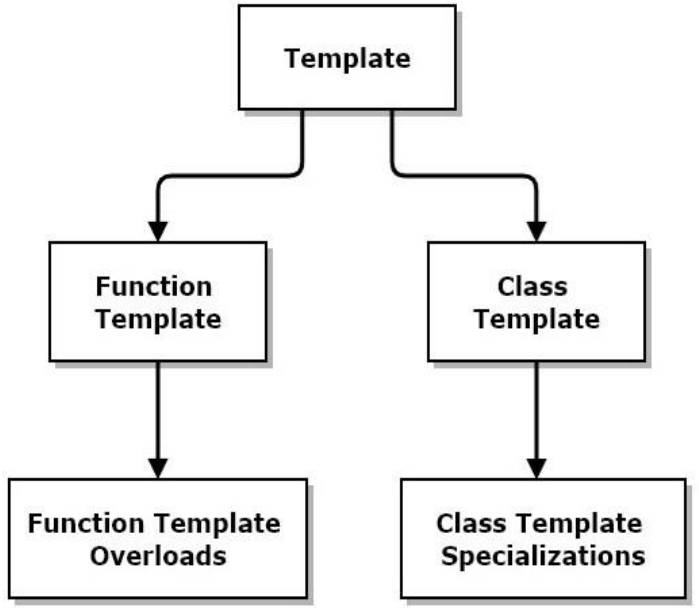
https://jimfawcett.github.io/CppStory_Models.html#templ

- Function and class templates are code generators that create functions or classes from parameterized patterns.
- Function templates generate concrete functions and class templates generate concrete classes when supplied, in application code, with specific types.
- The class template, shown top right, generates a point class for each T specified in application code. Here, the using code has instantiated it with two types, int and double, resulting in Point<int> and Point<double>, two distinct classes.



Template overloads and Specialization

- Template specification provides a pattern for generating specific code for a function or class.
- That gets instantiated by something close to substitution, subject to type deduction for classes or overload resolution for functions.
- That may work well for most of the types an application uses but may fail for one or more specific types.
- In that case, a function may be defined for specific type(s) that has modified code. C++ guarantees that the overload will be chosen if the application supplied type matches the overload.
- Similarly, a class may be defined for specific type(s) that has modified code. The language guarantees that the specialization will be chosen, instead of the generic class, if the application type matches the specialization, using template type deduction.
- A designer may provide any number of overloads and specializations as needed by the application.



Conclusions

- If you understand the 7 models, we've covered, I think you will find C++ syntax and semantics to be consistent and sensible.
- Some particular parts of the language discussed in the C++ Story but not here are intricate and require some study to master:
  - Template type deduction
  - Function overload resolution
  - Template metaprogramming
- But template type deduction and template function resolution just seem to work without deep analysis most of the time.
- Template metaprogramming is used largely be library developers, but that is getting easier to use with each new version of the C++ standard.

Epilogue

- In this presentation we've discussed models for:
  1. Code Structure
  2. Compilation
  3. Program Execution
  4. Program use of Memory
  5. Classes
  6. C++ Objects
  7. Templates
- We've focused on models and ideas, not the details of design and syntax.
- You can find an extended discussion of C++ in the CppStory:
  https://JimFawcett.github.io/CppStory_Prologue
- There is a lot of sample code for the CppStory:
  https://github.com/JimFawcett/CppStory
- With documentation:
  https://JimFawcett.github.io/CppStoryRepo.html