C++ a more in depth look into the language

Who am I?

- Martin Nestorov
- Junior 2nd sem.
- I like to type . . . a lot
- email: mdn150@aubg.edu
- Twitter: @mnestorov

Why I chose to make these small tutorials

- People seem to have some problems with the transition from C++ to FDS.
- I want to help!
- I want to also learn!

Over the course of the past several years I found a certain love for C++, but instead of just reading about the language and doing some small experiments with it, I wanted to share my knowledge and to get better at it. Seeing how some students struggle with C++ in the beginning courses, I decided that I can help out. That way not only am I learning more in-depth concepts, but also I am making the lives of students easier (hopefully).

How to look at C++

C++ as **four** sub-languages

- C
- OOP
- STL
- Templates

"C++ would make a decent teaching language if we could teach the ++ part without the C part." — Michael B. Feldman.

Bonus joke: "In C++ it's harder to shoot yourself in the foot, but when you do, you blow off your whole leg." — Bjarne Stroustrup.

C++ can sometimes be vary annoying, because we have absolutely no idea how it works and what it does. The fact that having very similar syntax for wildly different things doesn't help either. So instead of just banging our heads until bleeding sets in, we can try to understand a little bit better what is happening with this language and (hopefully) try to minimize **undefined** behavior. In fact C++ is that special language where if you don't know what you are doing, the language re-enforces that by acting in a way, where it doesn't know what it might do. That's pretty scary. That's why I will try to minimize this effect with these tutorials and help you out to make better C++ programs and to introduce effective methods for writing C++ code.

C++ can be generally divided into four sub-categories or sub-languages. It's a daunting task to go over all of these aspects and trying to teach everything is something I can't do! But instead I will try to cover some topics that haven't been covered in depth in the C++ lectures and are somewhat elude to the students. Now our list seems like this

What we are going to cover

- File structure
- OOP
- Inheritance
- Templates

- STL
- Resource Management
- Good practices
- Compilation
- Linking

File structure

- Advantages:
  - Better code organization
  - Faster compiling
  - Separation of **implementation** from **interface**
- Disadvantages:
  - More complicated structure

It has come to my attention that many people, while writing their homeworks, neglect the advantages of structuring their **source** files. This is most noticeable when they are writing the first 3 FDS homeworks or their FDS project. Being able to properly manage your project into several files and to navigate between them in an optimal and cohesive way will minimize your C++ suffering (and coding experience in general).

Now most of you already know this, but there are several great advantages to separating your **implementation** (cpp) files from your **interface** (h) files. One advantage is that this will bring code clarity to your program. Instead of navigating through one big **400** line `Source.cpp` file, and wondering where that one definition of that object is and then going back to the `main()` method to fix one line, you will have a nicely structured system. Every class will have it's own implementation and interface file, thus minimizing code searching and optimizing code writing! But that's not all. This separation will allow you to compile your programs faster. That is, every **source** file is compiled on its own. When you make a small change in your original 400 line monstrosity, you have to wait for all of that code to **re-compile**, but if you have the files separated, you just have to re-compile only the changed files, making waiting time much smaller (naturally this effect can be seen when compiling large programs, while small ones won't have such noticeable impact on time). And lastly, we also gain the benefit of implementation from interface separation.

But what we are sacrificing for these benefits is that we are increasing our project complexity. This shouldn't scare us that much, but we do have to keep this in mind - **We have to limit ourselves not to over layer our program**.

Now I am not going to tell you how to add files and where to put them, I will just show you how to properly implement header and source files and how to avoid collisions.

First things first. What's the difference between headers and sources? Apart from the obvious name difference, header files are considered to be the **interface** part of the source code, that is, the place where we explain what the program does, but **not** how it does it. The source (cpp) files are the **implementation** part, this is where we explain **how** we implement the interface. Keeping them separate allows us to keep a clean directory and structure and to separate our logic from our implementation.

How to work with headers?

```
0  #include <iostream> //header we always include
1  #include "awesomeclass1.h"
2  #include "awesomeclass2.h"
3  #include "awesomeclass3.h" // our own class headers
4
5  int main()
6  {
7      awesomeclass1 ac;
8      awesomeclass2 ac2;
```

```
 9      awesomeclass3 ac3;
10
11      ac.do_something(ac2);
12
13      ac3 = ac;
14
15      return 0;
16  }
```

So what are header files and what do they do?

Header files, usually ending with the `*.h` file extension (can be also `*.hxx`, `*.hpp`, or nothing at all), are code files that are **included** in our source files. We can look at header files as code that only serves as the interface to our classes. That is, it holds a declaration which we later on implement and/or include in our source files. Although we will talk about header files and how we `#include` them later on, what we can say now is that for every header file that is included, the contents of that file are put in the place of the `#include` keyword. What we can expect header files to do is to hold **declarations** inside of them. They just tell us what we are going to implement, but not **how** we are going to implement it (that's the job of the source files).

NOTE: that for the `iostream` we used angle brackets and for our classes we used quotes. This just means that the header file **iostream** comes with the compiler and that the linker should look in the system directories for the definition. The quotes indicate that we are providing the header file and that the linker should look at the build directory.

Header guards

```
 0  // x.h
 1  class x { . . . };
 2
 3  // a.h
 4  class a { X x . . . };
 5
 6  // b.h
 7  class b { X x . . . };
 8
 9  // source.cpp
10  #include "a.h" // include x.h for the first time
11  #include "b.h" // error! include x.h again!
```

```
0  // x.h
1  #ifndef __X_H_INCLUDED__
2  #define __X_H_INCLUDED__
3
4  class x { . . . };
5
6  #endif
```

Because we are going to be using a lot of headers in our program, we would need to include all of them. But because these files are actually copied when we **include** them, we would have several problems. One would be the code duplication. Another one will be the multiple declarations of classes and functions. To overcome this problem we would need to **guard** our header files in order not to use them more than once in our program. That is why we use the directives `#ifndef`, `#define`, and `#endif`. When we wrap our header file around with these directives, we ensure that the preprocessor will not copy a single file more than once into the source file.

Declarations vs Definitions

```
0  // declarations
1  extern int a;
2  extern char b;
3  class foo; // extern not allowed for user types
4  double sum(double, int);  // no need for extern
5
6  // definitions
7  int a; // the implementation of a
8  char c; // the implementation of c
9  double div(double a, int x) { return a / x; }
10 class foo { . . . };
```

I want to take some time now to revisit two old concepts before we continue - namely I want to talk about **declarations** and **definitions**. Here we will mention the role of the **compiler** and **linker** briefly, but we will cover them later on. Now a **declaration** introduces an identifier and its type, be that type of a variable, object, or a function. Basically we are saying to the compiler that there is something with this name and this type, but we are not specifying **how** this specific thing is implemented. This is what the **compiler** needs in order to make a reference to that declaration. We can imagine the situation where we have a function which we want to use in multiple files in our program, it's just a very useful function, but we don't want to provide the implementation of the function in every source file. That's why we just declare the function, the compiler sees the declaration, makes a reference to it without the need to know its implementation, and then when we build the final executable, the linker links every reference of the declared function to its implementation. If you remember from the C++ course, a function prototype is a declaration.

A **definition** in C++ is when we a define something in its complete form. This means that we need to both declare something and define it at the same time. We actually implement that specific identifier we have declared. When we write functions and give them a function body at the same time, that is a form of definition. When we write `int x;` we are both declaring and defining the variable `x`. This means that we are telling the compiler that there is a variable `x` of type `int` that is in the global scope in the current source file. It's important to note that giving this variable a value is not necessary in order to define it. But in the examples we have this specific keyword `extern` that is associated with declarations. What `extern` does is that it marks the variable and tells the compiler that this is only a declaration and that a definition will be found somewhere else. In general we use `extern` when we want to declare global variables in header files and **define** them in some source file.

We can **declare** something as much as we like in as many files as we want, but we can only **define** something only **once**. Often when we get errors telling us that something is undefined (this is at linker time), what that means is that we have something declared, but we have not defined its implementation anywhere and we get a **missing symbols error**. On the other hand when we have multiple definitions of the same thing, the linker is confused as to which specific definition we want and we get a **duplicate symbols error**.

Okay now let us go back to header files.

How and when to use headers

`class A` > (uses) > `class B`

- do nothing if: A makes no references at all to B
- do nothing if: The only reference to B is in a friend declaration
- forward declare B if: A contains a B pointer or reference: `B* b;`
- forward declare B if: one or more functions has a B **object/pointer/reference** as a parameter, or as a return type: `B MyFunction(B myb);`
- `#include "b.h"` if: B is a parent class of A
- `#include "b.h"` if: A contains a B object: `B b;`

```
0  #ifndef __MY_CLASS_H_INCLUDED__
1  #define __MY_CLASS_H_INCLUDED__
2
3  // Forward declare dependencies
4  class F;
5  class B;
6
7  // Include dependencies
8  #include <map>
9  #include "c.h"
10
11 // Our current class
12 class MyClass : public c
13 {
14 public:
15   std::map<int, char> matrix;  // a map object that is required by our class
16   F* f;                        // F pointer, so forward declare F
17   void Func(B& b);             // B reference, so forward declare B
18
19   friend class MyFriend;       // friend declaration is not a dependency
20                                // don't do anything about MyFriend
21 }
22 #endif
```

At this point, after we are revised what declarations and definitions are, we can talk about how to use headers and in which situations this is a viable option. The very short answer is - use header files when you have a class, a grouping of functions that do similar things, or a grouping of global variables. In general this approach is not bad and will fair1ly well for you, but we shouldn't stop here. That's just the surface of it, to dig deeper we have to understand in which situations we can use header files and when to use forward declarations.

Let's imagine that we will be writing a lot of classes, where there will be a lot of dependencies. So when we talk about header files and using multiple classes we are also talking about dependencies between classes (this may also mean just a grouping of functions, but for simplicity reasons, we will refer to a header file as being a single class that does one thing). As an example, a derived class will always be dependent of its parent class, which means that we must be aware of the parent class at compile time, so we can use the derived class (such a case will be when we have a **public** inheritance between two classes). There are two general forms of dependencies - one is, that there will be things that can be **forward declared**, and the other is, that things need to be `#include`-ed. Let's take an example with the following. Image we have class `A`, which uses class `B`. This makes A dependent on B. To decide if we need to **forward declare** or to include A depends on what the interaction between the two classes is.

Some explanation is required here. First of we are protecting our class with the header guards. Then we are forward declaring two classes `F` and `B`. This means that we are telling the compiler that there is a user type class called **F** and **B** and that the implementation of the classes are somewhere else. This allows us to access the methods and variables of **F** and **B** without needing to specify their implementation. Forward declarations can be useful, because they can reduce compilation time. If we just want to access and use one or two functions from a class, instead of including the whole class (and copying its contents to the current file) we can just forward declare it and save ourselves some time.

In general we want to remember this simple rule - prefer forward declarations to includes, in order to save compilation time and circular dependency issues. The benefits to these rules is that we are encapsulating the classes we use as much as possible, because the classes that use our classes don't need to know how they work. This means that if we restrict our use of includes and think about how the dependencies of the classes interact, we are writing encapsulated and decoupled code.

Circular Dependencies

```
0  // b.h
1  #include "a.h"
2  class B { A* a; };
3
4  // a.h
5  #include "b.h"
6  class B { B* b; };
```

I mentioned at some point about circular dependencies, but they are something that is very common when we are writing OOP code. In this example we see a circular dependency. If we were to just include both header files, we would be stuck in a loop, trying to figure out which class to include first, giving us a compilation error. In order to overcome this (and keep in mind that there is nothing wrong with such circular dependencies) we can just forward declare one of the two classes and include the other so we can break from this cycle. Now before you ask "what about the situation where we have explicit object declarations instead of pointers to objects?", let's just say that such a situation is **impossible**. Not that you can't write such a program, but there is an inherit design flaw to such code, it's virtually impossible to make such a code work, and there is a clear flaw in logic if you find yourself in such a situation. The other question you might have is - well when should we use pointers to objects? This totally depends on your program. For instance, if we have a `class Car` and a `class Wheel`, it's logical for the **Car** to include the **Wheel** class, but the **Wheel** class only needs a pointer to the Car object (it doesn't need the whole Car).

The ++ in Cpp

- C with classes
  - Classes and Objects
  - Constructors and Operators
  - Designs
- Virtuals and Inheritance
  - Multiple inheritance
  - Virtual functions and destructors
  - Inheritance models

When C++ first came out it wasn't all that different from C. Apart from having standard object orientation functionalities and inheritance capabilities, the two languages were more or less the same. But overtime C and C++ grew to become very different. We already looked at how to structure out program in terms of files and how it executes. Now we will dive into the world of OOP in C++, we will revise what was though in the C++ course and will try to build upon that material by giving examples, by running code, and talking about practices for writing better C++.

First we will cover classes. What the classes implement in the background, what to do with constructors, destructors, and operators. We will look over some designs on how to build our classes for maximum utility and efficiency. Then we will go over inheritance and polymorphism, with discussions over the different models, how to use multiple inheritance, what virtual tables are, how to use virtual functions, and so on.

C with classes

Classes and Objects

```
0  class Person
1  {
2  private: // (Used by default if not specified explicitly) No one can see in here
3      int age;
```

```
4
5       std::string name;
6       std::string job;
7
8       static const Doctor *d;
9
10      const void wave();
11
12  protected: // Can be seen only by class that is a child
13             // Usually we don't have a use case for protected
14      bool is_married;
15
16  public: // Everyone can see this part
17      Person();
18      Person(std::string name);
19      ~Person();
20
21      std::string get_name();
22      std::string get_job();
23      int get_age();
24
25      double calculate_salary();
26
27      bool get_occupation() { return is_married; } // implicit inline function
28  };
```

This is a basic class. We can see the scopes of the class and what we can have as variables and functions in the different places. A quick note here - although we have 3 scopes, the ones we use the most are `private` and `public`, while `protected` is used in seldom cases, so we need not worry about it that much.

Initialization lists

```
0   class Library
1   {
2   public:
3       Library(const std::string& name, const std::list<Books>& books);
4   private:
5       std::string theName;
6       std::list<Books> books;
7       int addressNum;
8   };
9
10  Library::Library(const std::string& name, const std::list<Books>& books)
11  // <- Initialization start right here before the opening bracket
12  {
13      theName = name; // these are all assignments
14      theBooks = books;
15      addressNum = 0;
16  } // This is OK but can be done better
```

Constructor with initialization list

```
0   Library::Library(const std::string& name, const std::list<Books>& books)
1   : theName(name), // These are all now initialized
2     theBooks(books),
```

```
3     addressNum(0)
4  { } // Empty constructor body
```

When we are initializing this object, we would write `Person p1;`, but what would happen with its data members. Will they be initialized to zero or not? We can never be sure about this and sometimes they might be **zero**, sometimes they might be left **uninitialized**. If we take the risk and read something that is uninitialized, then we will get undefined behavior. There are rules to remember when something is initialized to zero and when something is left uninitialized, but this all depends if you are writing the C part of C++ or the STL part, and things just get way to complicated. That is why we can take the safe approach and just **always** initialize our object when we create them. If we are just working with normal variables in a function, this would be like writing `int x = 0; double b; std::cin >> b; char* x = "pointer intialization"` and so on. For objects, however, this tasks is left to the constructor. The rule is simple - make sure that everything is initialized when an object is created.

In the example we can see that the Library constructor makes 3 assignments to the data members. In C++, the **initialization** takes place **right before** we get into the constructor body! This means that in order to make sure that the variables are initialized, we have to put them into an **initialization list** (how intuitive!). There are several advantages to this approach. First we make sure that we won't have uninitialized data members and will avoid undefined behavior. Secondly, because we are assigning objects (the `string` and `list` members) we are avoiding calling their assignment constructors. In the previous code, where we assigned the variables, first we had to initialize the `string` and `list` variables, then we had to assign them the new values by calling a copy assignment operator. That's way too much work. But with the initialization list, not such thing needs to happen. The name and the phone are **copy-constructed** from the passed parameters. This means that only a copy constructor of the `string` and `list` is called in order to initialize the data members, and this operations (the copy constructor) is much more efficient than the previous code.

If we want to initialize the data members to nothing or we don't have any parameters to pass to the constructor, we can just call the initialization list and leave the brackets empty.

There is one little aspect we must note here and that is - we must write the order of our initialization list in the **same order** we declared our data members in our class. This is just how C++ works, we must first initialize `theName` and then `theBooks` and then `addressNum`. If we don't do that we will get an error.

Static non-local objects defined in different translation units

```
 0  // in the filesystem.cpp file
 1  class FileSystem
 2  {
 3  public:
 4      std::size_t numDisks() const;
 5  };
 6  extern FileSystem tfs;
 7
 8  // in our .cpp file
 9  class Directory
10  {
11  public:
12      Directory(params);
13  };
14
15  Directory::Directory(params)
16  {
17      std::size_t disks = tfs.numDisks(); // we use the tfs object,
18                                          // but we cannot be sure
19                                          // that it is initialized
20  }
```

```
21
22  Directory tempDir(params);
```

The solution:

```
 0  FileSystem& tfs()         // this replaces the tfs object; it could be
 1  {                         // static in the FileSystem class
 2      static FileSystem fs; // define and initialize a local static object
 3      return fs;            // return a reference to it
 4  }
 5
 6  class Directory { ... };
 7                  // as before
 8  Directory::Directory(params)             // as before, except references
 9  {                                        // to tfs are
10      std::size_t disks = tfs().numDisks(); // now to tfs()
11  }
12
13  Directory& tempDir()               // this replaces the tempDir object; it
14  {                                  // could be static in the Directory class
15      static Directory td( params ); // define/initialize local static object
16      return td;                     // return reference to it
17  }
```

One final thing that is worth mentioning is the case when we have to **initialize non-local static objects from different translation units**. What that jumble of words means is that - if we have a **static object** (an object that has a duration from the beginning of its initialization till the end of the program) and if that object is **non-local** (it's in the global scope, namespace scope, or in the scope of an other class) and if it's in a **translation unit** (basically it's a single source file) how do we initialize it? The problem here is that if we use such a static non-local object, we can **never ever** be sure when it is initialized properly and we are down the road of undefined behavior. The way to overcome such situations is by writing a function that makes that static non-local object a static local object (basically implementing the **Singleton** design pattern. . . almost). We just need to write a small function in our code, where we define and initialize a static local object of what we need and return a reference to it. This way we make sure that every time the function to get the object is called, the object will **always** be initialized.

Constructors and Operators

```
 0  // Empty class
 1  class A {};
 2
 3  // Empty class again
 4  class A
 5  {
 6  public:
 7      A() {...}             // constructor
 8      A(const A& rhs) {...} // copy constructor
 9      ~A() {...}            // destructor
10
11      A& operator=(const A& rhs) {...} // copy assingment operator
12  };
```

All of the classes we write have at least several constructors in them which will be either user defined or put in the class by the compiler. Sometimes we don't have to explicitly tell C++ what kind of constructors we want, because our program doesn't requite such functionality, but sometimes we have to implement our own constructors and operators (**operator**

overloading). It's a good thing to know what the compiler is implementing instead of us when we are not writing the constructors ourselves.

Let's take this example for instance. Here we have a class that has nothing in it and we have a class that has four constructors implemented. There is virtually no difference between writing the empty class without constructors and the one with the 3 ctors and one operator implemented. The 3 cotrs and 1 operator that we will always have by default are - a default constructor, default destructor, copy constructor, and a copy assignment operator. In general there isn't much difference between what you will write as an implementation and what the compiler will write for there things, but keep in mind that in some specific cases you will be forced to write your own constructors. This will be the case when you have some constants and/or some references in your class. Why references? Because when you are copying an instance and you have a pointer in it, you would want to copy what the pointer points to (its content) and not the pointer only. And the case for `const` is that you would want to keep **const correctness** when copying. This is not a problem if our class has normal data members.

```
0  template<typename T>
1  class NamedObject
2  {
3  public:
4      NamedObject(std::string& name, const T& value);
5  private:
6      std::string& nameValue;
7      const T objectValue;
8  };
```

In this case when we have a reference, we just have to make sure that we are implementing a **deep** copy when we are writing the constructor or operator. But if our class has a `const` then we are in trouble. Because we cannot change the value of the initial object we want to re-assign, we cannot implement a proper copy assignment operator. In the case with the constant data member, a default copy assignment operator will not be implemented, and if we were to implement one of our own, we would not copy the `const` data members.

Designs

Write OOP classes

```
 0  class AccessLevels
 1  {
 2  public:
 3      int get_read_write() const { return readWrite; }
 4      int get_read_only() const { return readOnly; }
 5
 6      void set_read_write(int value) { readWrite = value; }
 7      void set_write_only(int value) { writeOnly = value; }
 8  private:
 9      int no_access;    // no access to this
10                        // (maybe it's used as an internal constant)
11      int read_only;    // only read access
12      int read_write;   // read and write access
13      int write_only;   // write only access
14  };
```

Although C++ gives us a lot of flexibility with what we can put into a class and under what type of scope, there have been several great designs that usually make our work much easier and allows us to quickly create our classes. This doesn't mean that we have to neglect what we are actually writing, but for most of the time we can follow these conventions.

First of - declare your data members as private (not even protected). Okay so why do this? First off, by making our data members private we are limiting the access to them, we can only **get** them and **change** them only through functions. This

allows us for finer control over our data and will also follow some consistency when writing classes. We won't need to bang our heads whether we are changing something or we are writing to something or whatever. The general rule is - every data member should be hidden, or - show only as much as you need. This means that some of our data members won't even need `getters` and `setters`, thus reducing our chances of bugs and lines of code. The last, and biggest, benefit to hide your data is **encapsulation**. This brings a lot of flexibility of implementation to the table and we should strive to make our classes as flexible and encapsulated as possible. A rule we can remember about this is - the more a class is used, the more it needs to be encapsulated. The same rules apply for `protected` data members. It might seem that they are encapsulated on first glance, but think about this - if we are to remove a **protected** data member, we are going to break an **undefined** amount of inherited classes, which is always more than what we will break if we just had that member as **private**. Seldom are the situations where `protected` will save the day.

In order to access are already `private` variables, we must rely on getters and setters. These are small and fast functions that do what their name suggests. It's usually a good practice to make you getters `const`, this way you are ensuring yourself that the variable won't change upon retrieval. As a reminder - not all data members need such functions to them!

Another thing that is worth mentioning about how to write classes is - do not over complicate them with functions that they **don't** necessarily need. This means that if a class needs helper functions, it's not necessary to put those functions as class methods. Although our intuition tells us that we need to encapsulate everything, this doesn't always mean - put everything into a class. Let's take a this source file for instance (`./src/robot_and_mines.cpp`). This is a little robot that goes through a maze, based on commands given by the user. To load the maze we use a function that reads a text file. The question is - since we are loading something specific that the robot will do, i.e. go through a maze, should that loading function be part of the robot class? Well maybe, it's not wrong to think that a robot might have the functionality to load its course. But at the same time maybe it's not such a good idea, because this increases the **size** of our class, we have another state to worry about, and if we were to extend this program even further, we might need to remove this function, since it will be getting in the way. So instead of making this function a class method or a friend function, just use it as a standalone thing that just helps the class. If we have multiple functions that are not really class specific, but are very helpful, we can just put them in a standalone `header` file or a `namespace` and used them from there. The takeaway is this - prefer to write non-member and non-friend functions to member functions (do not bloat your classes).

Use inline functions

```
0  class Person
1  {
2  public:
3      int age() const { return theAge; } // implicit inline request
4                                         // defined in class definition
5      friend job() const { return currJob; } // friend functions can
6                                             // also be inline
7  private:
8      int theAge;
9  };
10
11 template<typename T>                              // an explicit inline
12 inline const T& std::max(const T& a, const T& b) // request: std::max is
13 { return a < b ? b : a; }                         // preceded by "inline"
```

Inlining functions are one of the greatest ideas ever - they allows us to write simple and fast code! Well what does this exactly mean? Inline functions are those that are either specifically written in the class body or are marked as `inline`. This removes the function call and replaces it with the body of the function directly. This reduces the work of the compiler to jump to function definitions and in general, optimizes our performance. Another optimization is that if we have a code stretch that doesn't have function calls, the compiler might optimize this section, and if we inline our functions we might promote such compiler optimizations. Great for us! But there is a "but" in this whole situation - using too much `inline` functions will bloat our object file making much bigger executable. On machines that don't have a lot of memory won't handle this very well. **Inlined** code can lead to additional paging and **reduced** cache hit rate, and all of the negatives accompanying this. But if the code we inline is small enough, this will lead to smaller code for the function in comparison

to the code generated for the function call, which will **increase** cache hit ratio and will make object files **smaller**. So we must be careful what we inline and what we don't (duh).

We have to keep in mind that **inlining** is a request to the compiler and not a command. We can have two approaches, one is to make **implicit inline requests** by inserting the body of the function in our class definition. These functions are usually member functions, but they can also be friend functions.

Okay so when should we use inline functions? Well the thing is that the compiler will decide which functions to inline in reality. Inlining **in general** is a compile time thing (some build environments execute inlining during linking and some even during runtime, but the standard is during compilation). This means that we should put our `inlines` in header files. Let's look at another aspect of inlining and that is the fact that the compiler decides when a function should be inlined. Well in general, a function that has too complicated logic, like loops, recursion, and so on, won't be a good candidate. Also unless a virtual function is trivial simple, it too won't be inlined by the compiler. All of this depends on the build environment and on the compiler you are using, but luckily for us, we will get warnings when a inline function might not be suitable, so that is a good indicator whether we are doing something correct. A good approach is to inline **nothing** in the beginning, then go over your code again and see what can be optimized further.

Treat class design as type design

- How should objects of your new type be created and destroyed?
- How should object initialization differ from object assignment?
- What are the restrictions on legal values for your new type?
- What operators and functions make sense for the new type?
- What standard functions should be disallowed?
- Who should have access to the members of your new type?
- Is a new type really what you need?

One of the things that will happen when you are writing your FDS project is that you won't have an idea what to do in the beginning and how to start. Hence you will hesitate how to build your classes and what to do with them.

Writing classes isn't easy, because in C++ we are not creating only a class, but we are also creating a **user type**. This means that we have to take care of what that type must to do and what it must **not** do. That's why we can ask ourselves the following question in order to get a better grip of the classes we are writing. In consequence we will get a better understanding of the program we are writing.

How should objects of your new type be created and destroyed? - How this is done influences the design of your class's constructors and destructor, as well as its memory allocation and deallocation functions.

How should object intialization differ from object assignment? - The answer to this question determines the behavior of and the differences between your constructors and your assignment operators. It's important not to confuse initialization with assignment, because they correspond to different function calls.

What are the restrictions on legal values for your new type? - Usually, only some combinations of values for a class's data members are valid. Those combinations determine the invariants your class will have to maintain. The invariants determine the error checking you'll have to do inside your member functions, especially your constructors, assignment operators, and "setter" functions. It may also affect the exceptions your functions throw and, on the off chance you use them, your functions' exception specifications.

What operators and functions make sense for the new type? - The answer to this question determines which functions you'll declare for your class. Some functions will be member functions, but some will not.

What standard functions should be disallowed? - Those are the ones you'll need to declare private.

Who should have access to the members of your new type? - This question helps you determine which members are public, which are protected, and which are private. It also helps you determine which classes and/or functions should be friends, as well as whether it makes sense to nest one class inside another.

Is a new type really what you need? If you're defining a new derived class only so you can add functionality to an existing class, perhaps you'd better achieve your goals by simply defining one or more non-member functions or templates.

Prefer pass-by-reference-to-const to pass-by-value

```
class Window
{
public:
    std::string name() const;      // return name of window
    virtual void display() const; // draw window and contents
};

class WindowWithScrollBars: public Window
{
public:
    virtual void display() const;
};

// the bad functoin
void printNameAndDisplay(Window w) // Bad! Parameter might be sliced!
{
    std::cout << w.name();
    w.display();
}

// the correct function
void printNameAndDisplay(const Window& w) // Correct! Parameter won't be sliced!
{
std::cout << w.name();
w.display();
}

// the call
WindowWithScrollBars wwsb;
printNameAndDisplay(wwsb);
```

Most of you are aware of what **pass-by-value** and **pass-by-reference** is, but here we will just discuss a few more advantages to using passing of references to `const`. One thing that happens when we are passing objects (and more specifically, **inherited** / **derived** objects from a base class) by value is that we are at the risk of **slicing** off the derived part of the object in the function call. What this means is that if we have a `class A` which inherits `class B`, and we pass an object of `type A` to a function by value (the function parameter accepts `type B`), the **A** part of that object will not be passed, and only the **B** part will remain. Okay maybe that sounds too complicated, let's revise. By passing something by value we are copying it and the function is using the copy. In the case of inherited classes, when we are passing a derived object by value the **copy constructor** of the class is called to make a copy - naturally. But when the object has a base class, the base class is copied first. If the parameter of the function is of the base class type, only the base class during the copying will be left, and the derived part will be omitted. That is why it's a better idea to pass-by-reference-to-const.

Under the compiler, **references** are passed as pointers, so we are technically just passing a pointer. That is why if we have a build-in type like `int` it is more efficient to pass by value. Also we can say that the STL **iterators** and **function objects** are also good to pass-by-value, because they have been specifically build to be more optimal to copy than to be passed as a reference (as a pointer).

Virtuals and Inheritance

- Multiple inheritance
- Virtual functions and destructors

- Inheritance models

Let's talk about inheritance. It is one of the central ideas behind OOP and C++ gives us a lot of freedom to express ourselves. We are able to have single and multiple inheritance, then each link can be public, private, or protected. Then we can have virtual or non-virtual linking. Member functions can also be virtual, or non-virtual, or pure virtual. These are a lot of states to think about and we have to make sure that we understand exactly how these things work together. We are going to talk about public and private inheritance, difference between inheritance of interface and of implementation, how and when to use virtual functions, when to use multiple inheritance, etc.

But first we have to be sure what virtualization means in C++ and how to use *virtual inheritance* and *virtual functions*.

Dealing with multiple inheritance

We already know how inheritance works, and although we know that there are several ways to do it, it's better to get an idea about multiple inheritance as well from the start. As we can see from the two pictures, we have an inheritance hierarchy.

```
 0  class BorrowableItem
 1  {
 2  public:
 3      void checkOut();
 4  };
 5
 6  class ElectronicGadget
 7  {
 8  private:
 9      bool checkOut() const;
10  };
11
12  class MP3Player:
13  public BorrowableItem,
14  public ElectronicGadget
15  { . . . };
16
17  // in some function
18  MP3Player mp;
19  mp.checkOut(); // ambiguous! a.k.a. problem!
```

The thing with *MI* is that it allows us to inherit the same name function from more than one base class. From the first source we can see how we have inherited two functions with the same name and thus we confuse the compiler which form of `checkOut()` to use. You might notice that the two functions have *different access* levels, but since this function call has to be resolved during compilation, it doesn't matter that the access levels are not `public` or `protected`. What we can do here is to explicitly tell the compiler which version of the function to use, but that would lead to more complications. One thing that this form of multiple inheritance can lead to is the *deadly MI diamond* (scary!).

```
1  class B {. . .};
2  class X: public B {. . .};
3  class Y: public B {. . .};
4
5  class Z: public X, public Y {. . .}; // Problem!
6                                       // 2 anonymous instances of B
```

```
1  class B {. . .};
2  class X: virtual public B {. . .};
3  class Y: virtual public B {. . .};
4
5  class Z: public X, public Y {. . .}; // When base class is inherit as
```

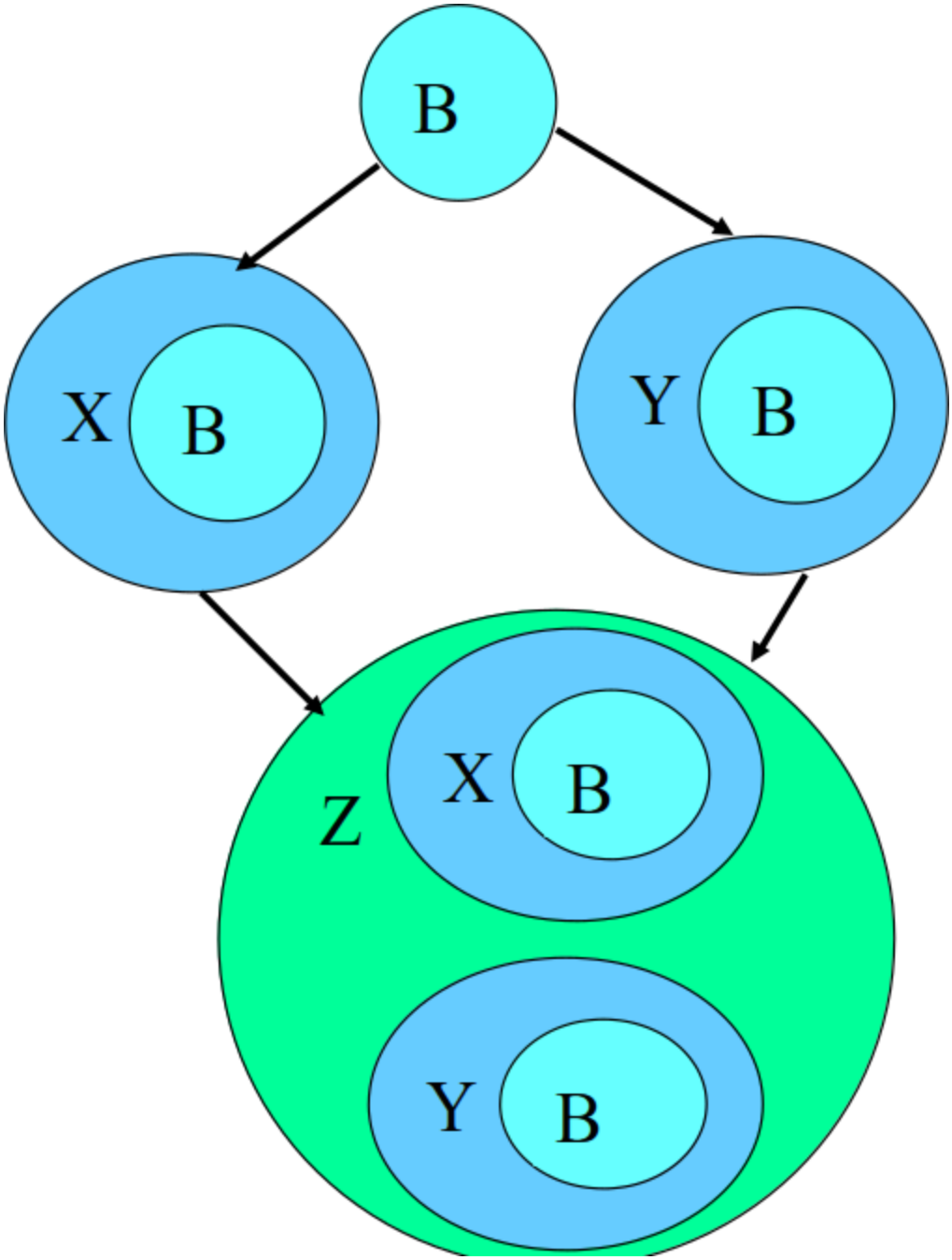

Figure 1: inh1

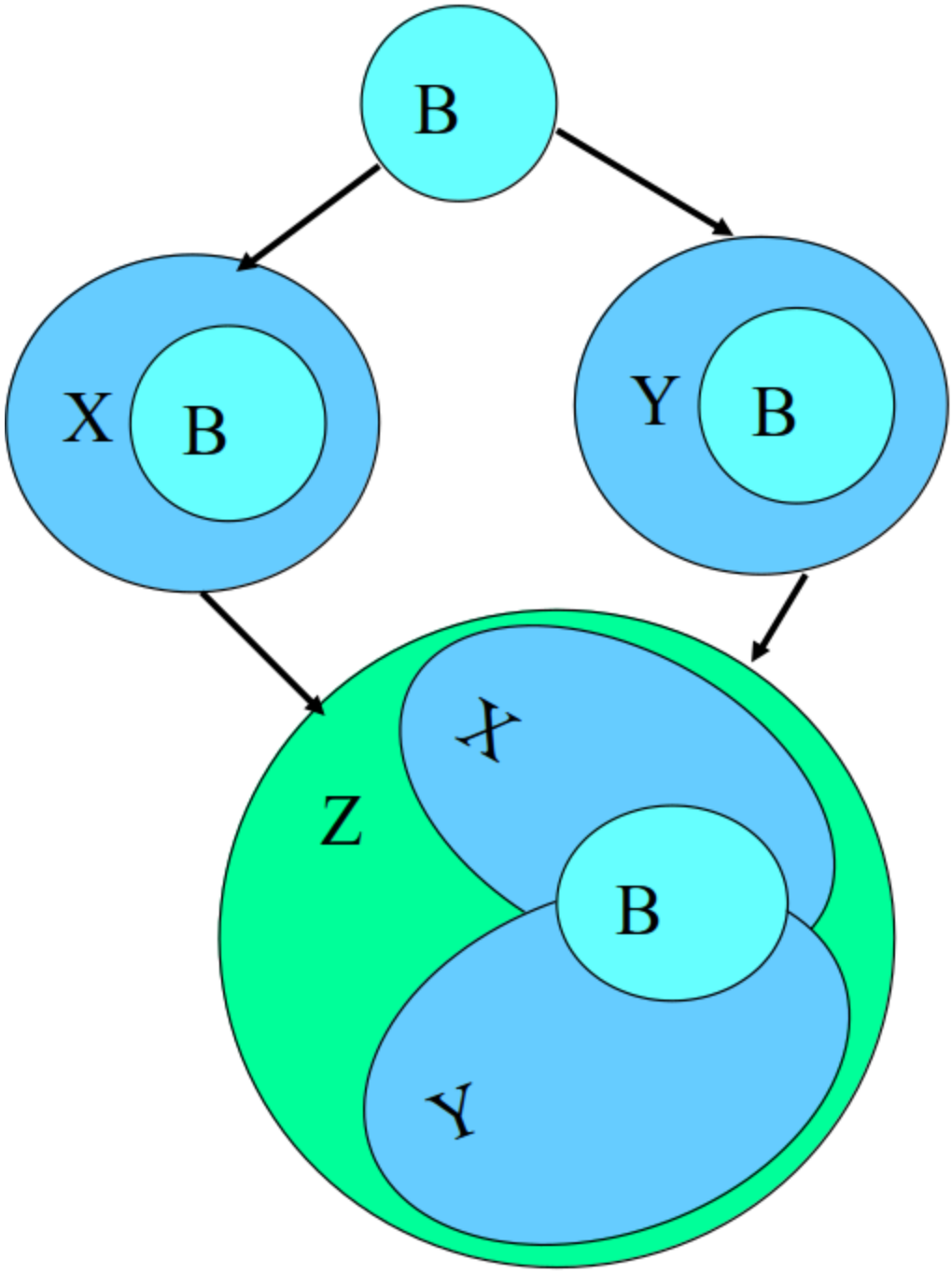

Figure 2: inh2

```
6                                       // virtual, only anonymous copy in
7                                       // the child class is created!
```

This means that when we have MI where the child class can have several paths to the base class, we have to ask ourselves, do we want the data members of the base class to be copied into two instances or to remain as one. By default C++ makes copies of the base class, but we have to option to rebuttal and to insist on having only one. Here is where *virtual inheritance* comes to the rescue. This means that we can have this diamond shape inheritance form and not suffer from data duplication simply by putting the `virtual` keyword when we are inhere ting from the base class. This will tell the compiler that there should be only one instance of the base class and in consequence we should have it only in one place in our *virtual tables*. Thus we are creating a *virtual base class* when we do this sort of virtual inheritance. The problem is that this will create bigger object files and will slow down our program (it causes overhead). So instead of just immediately writing virtual everywhere, here is some advice to follow when we are in such situations - *1.* Don't use MI unless you need to. By default use non-virtual inheritance. *2.* If you need to use virtual inheritance, try not to put any data members in the class. That way you are avoiding odd behavior when initialization takes place. Advice *2* is how technically `interface`'s work in `Java` and `C#`.

Virtual functions and destructors

Let's go over virtualization and bindings in C++. Now we have two types of function binding in the language - *static* and *dynamic*. *Static* binding means that when a function is called, the compiler knows which functions was called and jumps to it. This is the default behavior of C++. This, in consequence, allows us to have *function overloading*. Even though there are two or more functions with the same name, the compiler uniquely identifies each function depending on the parameters passed to those function. On the other hand, *dynamic* binding tells the compiler to match a function call during *runtime*. This is achieved through *virtual functions*. A base class pointer points to the derived class object. And a functions is declared virtual in the base class. Then the matching function is defined at run-time using a virtual table entry.

The virtual table is an interesting concept that allows us to have virtualization in C++. This table is a look up table of functions used to resolve function calls in a dynamic/late binding manner. This is also referred to as a *vtable* or a *dispatch table*. Now, every class that has at least one virtual function, or is derived from a class that uses virtual functions, has its own `vtable`. This table is simply a static array that the compiler sets up at compile time. The virtual table contains one entry for each virtual function that can be called by objects of the class. Each entry in this stable is simply a function pointer that points to the most-derived function accessible by that class. After that the compiler adds a hidden pointer to the base class. We can call this the `*_vptr`. This pointer is automatically set when a class instance is created so that it points to the pointer, which is actually a function parameter used by the compiler to resolve self-references. Keep in mind that this is in fact a real pointer. Consequently, it makes each class object allocated bigger by the size of one pointer. It also means that the pointer is inherited by the derived classes. This means that calling a virtual function is slower than calling a non-virtual one. Because we have to use the `*_vptr` to get to the appropriate virtual table. Then we have to index the virtual table to find the correct function call. Only then can we call the function. As a result we have to do 3 operations as opposed to 2 for a normal indirect function call, or 1 for a direct call.

Virtual Destructors

```
1  class TimeKeeper
2  {
3  public:
4     TimeKeeper();
5     ~TimeKeeper();
6  };
7
8  class AtomicClock: public TimeKeeper { . . . };
9  class WaterClock: public TimeKeeper { . . . };
10 class WristWatch: public TimeKeeper { . . . };
11
```

```
12 // some factory function in some source file
13 TimeKeeper* getTimeKeeper(); // returns a pointer to a dynamic-
14                              // ally allocated object of a class
15                              // derived from TimeKeeper
16
17 TimeKeeper *ptk = getTimeKeeper(); // get dynamically
18                                    // allocated object
19                                    // from TimeKeeper hierarchy
20 // some code that uses it
21
22 delete ptk; // release it to avoid resource leak
```

You probably have heard about *virtual destructors*. For most people they are some sort of a mystery, so now we will try to understand what they are good for and how to use them.

In the example we can see how we have multiple inheritance of a base class for time keeping. For ease of use we have a factory function that returns a derived class pointer (to some of the derived classes), but is deleted through a base class pointer. The problem is this - the base class `TimeKeeper` doesn't have a *virtual destructor*. C++ specifies that when a derived class object is deleted through a pointer to a base class with a non-virtual destructor, results are *undefined*. What happens is that, typically, the derived part of the object gets left out and doesn't get destroyed. This means that if we get an `AtomicClock` returned to us, only the base class will be destroyed, and will leave the whole object into a *semi-deleted* state. This cause data structure corruption, data corruption, and memory leaks. Not something we want. To overcome this, we have to declare the base class as a *virtual destructor* in order to tell the compiler to dynamically go to the destructor of the derived class so that the full object is deleted from memory. The general rule to follow is - if a base class has a virtual function, then it should also have a virtual destructor as well. If a class doesn't have a virtual function, then it probably shouldn't be used as a base class and shouldn't have a virtual destructor as well. The problem of making every destructor virtual is that it makes the whole class expand with one extra pointer that is not needed, which may cause code bloat and problems later on (especially if we want our code to be compatible with other languages). This is why we only declare virtual destructors when we have at least one virtual function in our class.

```
0 class TimeKeeper
1 {
2 public:
3     TimeKeeper();
4     virtual ~TimeKeeper();
5 };
6
7 TimeKeeper *ptk = getTimeKeeper();
8
9 delete ptk; // now behaves correctly
```

Inheritance models

- is-a
- Composition
- has-a
- is-implemented-in-terms-of

```
0 class Person { };
1 class Student: public Person { };
```

There are several inheritance models we can use. The same way we can have `public`, `private`, and `protected` as access modifiers, we can have the same model of inheritance between classes. But the thing here is that all of these mean wildly different things.

Let's start with `public` inheritance. If there is something that you absolutely **must** remember is this : public inheritance

means is a. This means that every time you write `class A : public B` you are saying to the compiler (and to other readers of code) that `class A` **is a** type of `class B` but **not vice versa**. **A** is just a more specialized form of **B** and **anywhere** B is used, A can be used respectively without a problem, but if you use a B in the place where A is needed, things won't work. This is the whole idea behind `public` inheritance.

If we take our example, we can clearly say that a every `Student` is a `Person`, but not every person is a student. This means that every data member and member function is `Person` can be used in `Student` and be correct, but we cannot do the reverse thing. This is intuitive enough and there isn't much to confuse us.

Note: `Private` inheritance on the other hand means something completely different and we are going to look at it in just a minute. `Protected` inheritance is something that doesn't really have a concept behind it (or at least the same way `private` and `public` have) so we will just not talk about it.

```
0  class Bird
1  {
2  public:
3      virtual void fly();
4  };
5
6  class Penguin: public Bird
7  {
8  };
```

Let's look at a simple example to get a taste for `public` inheritance. Here we can see that the class Penguin is a Bird. So far so good. But wait a minute. Penguins cannot fly, so we have a problem. We have to change something to make **public inheritance** work.

```
 0  class Bird
 1  {
 2  };
 3
 4  class FlyingBird: public Bird
 5  {
 6  public:
 7      virtual void fly();
 8  };
 9
10  class Penguin: public Bird
11  {
12  };
```

Here we fix the problem… kind of. OK but what if the program we are writing doesn't really need a `fly()` method or that we don't have the conception of implementing something that can fly. Does that mean that our original design was correct? Maybe it was. It all depends on what we have to do. This goes for show how our perception of things can easily misguide us. In general in software design there isn't one right way to do things, but we have to think about how we implement things. I want to stress the point that until now, most of us are used to make associations between real life objects without really thinking about them. But when it comes to implementing part of life into a program, we have to be vary judicious on how we do that. When I say that **public** inheritance means that `class A` is a `class B` completely, this means **completely**, we cannot close our eyes to just one thing and ignore it.

Inheritance of interface and implementation

```
0  class Shape
1  {
2  public:
3      virtual void draw() const = 0;
4      virtual void error(const std::string& msg);
```

```
5       int objectID() const;
6   };
7
8   class Rectangle: public Shape { . . . };
9   class Ellipse: public Shape { . . . };
```

Public inheritance is pretty useful, but we should think about how we use it and in what context. Here I want to talk about the several options we have. I have talked about interface and implementation quite a bit and in C++ we can inherit only the interface and provide our own implementation, we can inherit both the interface and implementation, but override the implementation, and we can also inherit both and not allow to override the implementation. Being able to make the difference between all of these things is quite important in order to understand the other models of inheritance we will cover.

In this example, we have three cases for what we need to talk about. We have a *pure virtual* function, a normal virtual function, and just a normal function in the base `class Shape`. Now apart from the fact that we cannot make instances of the base class, these three functions represent how to can interact with interfaces and implementations.

First let's look at the pure virtual function. Because we have the situation where we want to draw a specific shape, it is normal for our program to support such functionality. What the pure virtual function is telling is - we can draw whatever shape we want, but we have to say *how* to draw that specific thing. We do not have any default behavior so we are just inheriting an interface and only that. We can't make a generic algorithm to draw everything, so we are just getting these interface part of the class and implementing the logic ourselves.

Then we have the `error` function. It's just a normal virtual function, which means that it has some default implementation in the base class. This is the situation, where we are both inheriting the interface and implementation logic, but we are giving ourselves room to redefine the error logic if we want to. That is - it's normal for every shape to have a specific error message to show, but if we are in a situation where we cannot specify a custom error, we still want to output something to the user, and that is why having a default option to fall back on is very useful.

Finally we have the normal method that is getting the `id` of the object. Here it's pretty simple. We are saying that every object has the same logic of implementing the `id` function and we do not want to change that logic. So we are just declaring this function in the base class so that every object can have an id. This allows us to inherit both the interface and implementation and not alter anything from the inheritance.

Composition models

Now the last two models we should look at are the `has-a` and `is-implemented-in-terms-of` models. But they are somewhat different than *"simple"* public inheritance, and they mean totally different things. In short - *composition* is the relationship between types that arises when objects of one type have objects from another type. That is, we are modeling something that is *composed* of something else. The two models both use composition, but have different meanings in terms of the modeling part.

has-a is in the *application domain*. Meaning - when our program is modeling something, we are modeling real world things, like people, cars, games, screens, etc. This is the application domain. *is-implemented-in-terms-of* means that we are modeling implementations like data structures, buffers, mutexes, etc. This is part of the *implementation domain*.

has-a

```
0   class Address { . . . };
1   class PhoneNumber { . . . };
2
3   class Person
4   {
5   public:
6   private:
7       std::string name;       // composed object
```

```
 8      Address address;        // composed object
 9      PhoneNumber phoneNumber;// composed object
10      PhoneNumber faxNumber;  // composed object
11  };
```

Here are modeling something from the real world, where a `Person` *has-a* name, *has-a* address, *has-a* phone number, etc. The roles between *is-a* and *has-a* are very clear.

is-implemented-in-terms-of

```
0  class Person { . . . };
1  class Student
2  : private Person { . . . };  // inheritance is now private
3
4  void eat(const Person& p); // anyone can eat
5  void study(const Student& s); // only students study
6  Person p;  // p is a Person
7  Student s; // s is a Student
8  eat(p);    // fine, p is a Person
9  eat(s);    // error! a Student isn't a Person
```

Okay, we covered some different models and we know what `public` inheritance does. But what about `private`? What does it do and what does it mean? And how and when should we use it? All of these are valid questions and we will cover them now.

We can see from the example that we get different behavior when using *private inheritance*. Private inheritance works like this - the compiler will not convert a derived class into a base class and all of the protected (that's why the eat function fails at the end, because a `Student` is not a `Person`) and all members of the base class will be transformed into a private into the derived class. But what does all of this means? It means that private inheritance is a *is-implemented-in-terms-of* inheritance. If we have a `class A` which inherits from `class B` it does so because it is interested in the way some things in **B** are implemented and **not** because there is any conceptual relationship between the two classes. This makes class A into an implementation technique. That's why when we inherit privately, everything becomes private in the derived class, because we just want to use the implementation and not show it.

Now the question is this - when should we use `private` inheritance and when should we use *composition*? The general approach is this - use composition by default, and `private` inheritance when you must. The *must* part comes in when we have protected members and/or virtual functions.

Exercises

- Walk in minefield
- Todo list part 1
- Todo list part 2
- Get your priorities straight

References

File Structure

- How to use header files
- Header files
- Why do we have header and cpp files

- Declaration vs Definition
- Difference between declaration and definition
- The PIMPL idiom
- Forward declarations

Classes and Objects

- Copy constructors, assignment operators, and exception safe assignment
- The Problem with const Data Members
- Copy assignment
- Functions / Function Objects

Inheritance

- Multiple Inheritance
- Deadly MI Diamond
- Virtual inheritance