CSC A48 – Unit 6 – Designing and Building Good Software

1.- What this Unit is about.

So far, we have been thinking of problem solving in terms of the concepts, ideas, and algorithms we may need to use to get a particular task done. We should also spend some time considering the related issue of ***how to organize the software we're writing to solve the problem*** in a way that makes our solution as useful to others as possible.

In this Unit, we'll discuss the general principles that guide the organization of larger pieces of software (as opposed to small, self-contained programs such as the ones we've been implementing through the term). The goal is to understand how and why we need to think ***very carefully*** about the way we design the software we're writing, and learn about principles that have been developed to help us better organize, maintain, and expand on a given software solution.

The principles you learn here, will be the foundation on which you will build a full understanding of the software design process, a topic you will explore in depth in the Software Design (B07) course, and then refine and apply in the two software engineering courses (C01 and D01).

2.- Software as a collection of modules.

Up to this point, we have been using code that was written for us by someone else. All the functions in standard C libraries that you have used to implement your exercise solutions and your assignments had to be designed, developed, and tested well before they shipped with your compiler so you could easily have '*printf()'* or *'qsort()'* at your disposal, for example.

The question you should be asking yourself is: ***what would software look like if we didn't have any libraries for commonly used functions?*** If you think about it for a moment, you'll realize that the answer to that is not very nice:

Figure 1: Cartoon courtesy of Randall Munroe, XKCD, https://xkcd.com/1838/

Without some form of ***organization***, our software would be a big pile of code, with all kinds of functionality mixed in - there would be functions for carrying out the algorithms we need, but also all kinds of un-related things, like printing to the terminal, reading user input, managing files, implementing basic math, etc.

Anyone trying to understand our software, or worse, someone with the task of finding and fixing any existing bugs in our software would have a very difficult time dealing with our code.

So, we know that instead of writing large piles of code that does every kind of thing, we should instead ***figure out how to break down our application into a set of separate components*** each of which can be implemented as a ***module*** with its own code, and which can be understood, tested, and debugged without having to sift through the rest of the application's components (or with only minimal need to do so).

In effect, what we want is to build a large ***library*** of ***modules*** that we can ***reuse***. For example - we may want to write a module that sets up the adjacency matrix for a graph, and allows us to do path-finding on it using ***DFS***. But we want to write this module in such a way that we can use it ***in any application that needs to do path finding in a graph***.

This is not a trivial consideration - your software needs to be designed just right, so that it can work on a graph for any data you may want to explore, so that it can easily be called from software ***not written by you***, and so that it ***can be tested, debugged, and possibly expanded*** by other developers who don't know what your thoughts were when you originally wrote the code.

Let us look at the principles that should be in your mind when you start exploring the world of software design next term in B07.

3.- A wish list for the software we develop

If we are going to put our time and effort into developing software for solving a problem, we want to make sure the effort we put into results in the best possible software. Below is a list of properties that we should carefully consider when designing our software - not every single one will apply to every single problem, but we must always consider them all before we sit down to design and develop our solution.

**** Modularity*** - Our software is composed of separate ***modules*** each of which has ***one particular*** task, and each of which is mostly self-contained, can be understood, tested, and maintained independently of the others.

A well thought modular program will help:

- Reduce replication of code.
- Improve the chance that code you write will be reused.
- Make it easier to test your code and verify it is correct.

- Help you see the big picture of how your software is structured and how it works.

* ***Reusability*** - Writing good software requires a lot of thought and hard work, we want to ensure any modules we write for a specific task can be re-used by any other application that requires that specific task solved. For instance, if we develop a module that to find the shortest path between two nodes in a graph (which is a very common problem in many application domains). We want our implementation to be such that anyone needing to find the shorted path between graph nodes can take our module and build it into their application.

* ***Extendibility*** - We want our software to be easy to extend and improve. This allows us to build on software over time by improving and expanding its usability and functionality.

* ***Maintainability*** - Our software must be organized, easy to understand, well documented, and be free of unnecessary complexity. This improves our ability to test it, debug it, and upgrade it as needed over time. ***A competent developer not familiar with your code should be able to quickly get to the point where they can work on/with it.***

* ***Correctness*** - Any software we develop and release must have been thoroughly tested and made as close to bug-free as possible. Where appropriate, suitable tools should be used to determine correctness, code should have been reviewed by experienced developers not related to its implementation, and a suitable process must be in place for documenting, keeping track, and taking care of bugs found after the software is released.

* ***Efficiency*** - We have spent a good amount of time thinking about complexity and how to study the efficiency of our algorithms. We expect good code to be efficient both in terms of the algorithm chosen to solve a problem, and also in terms of how that algorithm is implemented (remember, at the end of the day, the constant terms will make a difference between different implementations of the same algorithm).

* ***Openness*** - When possible (e.g. when we're not developing software for a company that has a stake on what we develop), we should consider contributing to the open source software community. There is a lot of good work done for no other gain than to provide something useful for others. And we can contribute to this effort. Open-source software projects are a good way to make sure your work directly benefits others!

* ***Privacy and security*** - More and more, the software you write will be part of a system running over a network (possibly hosted somewhere else than the user's computer). Therefore it's important to pay close attention to the best

current practice in secure data exchange, as well as having a reliable solution for safely storing a user's personal information.

4.- How modules are organized and used in C

So far, we have been working with programs that contain only a couple program files at the most. We use compiler directives ('`#include`') to import all the code into a single program which then is compiled into our application's executable.

This is only suitable for very small programs, and for applications with very limited functionality. As we said above, we want to split complex applications into ***modules*** that implement part of the application's functionality. Each module will have its own program file(s), and they all need to be brought together in the right way to generate the executable program for our application.

In C, this is done by splitting each module into:

* A ***header*** file - these are files with extension '`.h`', and contain only the function ***declarations***, with ***no code***. You've already been using header files for common C libraries such as '`stdio.h`', and '`stdlib.h`', these provide the function definitions the compiler needs in order to know what to do when you call functions in these libraries, like '`printf()`', and '`calloc()`'.

* One of more ***program*** files - these have the extension '`.c`' and contain the actual code needed to implement the functions declared in the ***header*** file. You're already familiar with how this works. We only need to think about what to do when we have multiple program files for our application.

Applications are built from multiple such modules by

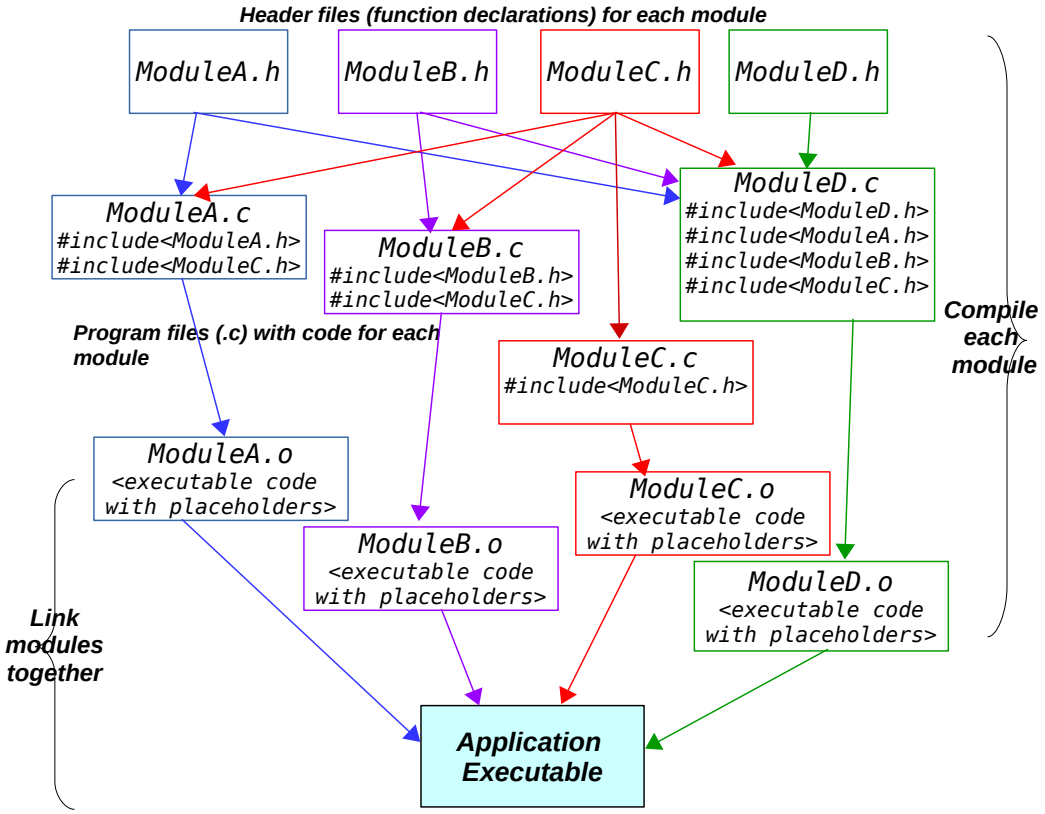
* Compiling each separate module into ***executable code with placeholders for functions from other modules.***

* ***Linking*** all the modules together: bringing together the code that implements the functionality from each of the modules, and updating the place-holders with the actual function calls to implemented function code.

The entire process is illustrated in the figure below. An application consisting of four different modules is split into multiple files - there are four header files, one for each module; and four implementation files (.c files with the code for each module). Note that each module may use functions from the others, and uses *'#include*' statements to bring in the function declarations for those modules for compilation.

Each module is compiled into a file (with extension '.o') that contains executable code with placeholders for the functions from other modules (you create a .o file by using the '-c' option when

calling gcc, this tells gcc that it's not expected to generate a full executable program from that module alone). Then all the .o files are ***linked*** together (at which point the placeholders are replaced with actual working function calls) and the complete application program is generated.



This is exactly the same process that happens when we include libraries of functions already provided with C (such as printf() from *'stdio.h*'). The compiler knows where the ***header file*** is located, uses the declaration of printf() provided there to know what to do when you use it with your program, and it also knows where the pre-compiled code for every function in *'stdio.h'* is stored so it can link it with the rest of your program and generate an executable.

The process above is cumbersome to do by hand if you have a large application, so, next year in B09 you will learn how to automate the process of building an application spread across several modules by writing a ***makefile*** that tells the compiler exactly what it has to do to compile each component and produce your executable program.

5.- Designing a good API

Back to the problem of how to build our software properly. The key problem that concern us is ***how will other programs interact with our module?*** that means we have to think about the functionality that will be *exposed* to code using our module, the various different functions that need to be provided, and the way in which information will be passed to and from these functions.

Setting down the details of how modules communicate and interact with each other is the job of the ***Application Programming Interface*** or ***API***. The API contains all the specifications needed to make use of, and to interact with, a software module, code library, subsystem, and even working components of an operating system or a remote server.

Here are some examples of commonly used APIs you may need to work with in the future:

- Google Maps : Allows you to make use of the Google maps framework for plotting locations and for finding paths between points in maps - *among many other things!*
  https://developers.google.com/maps/documentation/javascript/tutorial

- TensorFlow: Allows you to set up, train, test, evaluate, and operate a deep neural network for solving a task that requires learning from a very large dataset. Nowadays this is behind some of the most useful applications in A.I.
  https://www.tensorflow.org/

- Amazon AWS: Amazon's cloud-based AWS runs a large portion of internet-hosted services, and powers all kinds of applications from on-line trade to providing computing power for large simulations.
  https://docs.aws.amazon.com/index.html#lang/en_us

- Unity: Possibly the most popular API for creating, manipulating, and rendering 3D content, from graphical user interfaces and simulations, to interactive programs and games.
  https://docs.unity3d.com/ScriptReference/

The above is just a microscopic sample of the universe of APIs our there. Each of them is a world of complexity ***but the key is - we don't have to ever look at the code that implements them if we don't want to!***

The usefulness of an API is that it allows us to use a module, library, or service, without having to know the ***details*** of how it's implemented. All we need to know is how to pass information to the components of that module, and how to get back results. This will be easier or harder, depending on whether the API is well designed or not. A badly designed API will make software building cumbersome and reduce the usability of the component for which the API was designed.

In C, the API consists of the function declarations (in the .h) file, along with any constants and other important values defined there - it also includes all the documentation (at the top of each function) that describes what each of the functions does and their parameters and return values, and any documentation that is maintained externally (e.g. manual pages, a wiki, or a webpage describing the API and providing examples for developers).

In a sense, whenever we provide you with starter code for an assignment or exercise, we're defining a mini-API for you to use in completing a task - up to now you haven't needed to think about

just how we decided to implement specific functions in a certain way. Let's see what goes into designing a good API and why it is a challenging but important process.

Why thinking carefully about API design matters

Suppose you're writing a module for graph manipulation that supports finding path between nodes in the graph. The core function is implemented by this function:

```
intList *findPath(int Adj[N][N], int start, int goal)
{
   /*
      This function returns a linked-list of nodes
      that form a path from 'start' to 'goal' in
      the input graph described by the adjacency
      matrix Adj. 'start' and 'goal' are the indexes
      of the nodes we want to find a path between.
      If there is no path, the function returns NULL.

      Assumes: Adj is size (N*N) where N is the number
               of nodes in  the graph. Adj(i,j) is 1
               if there is a node from i to j, and 0
               otherwise.
               i,j are node indexes in [0,N-1]

      intList consists of nodes that contain:
               int nodeIndex;
               pathList *next;
   */
   .
   .
   .
}
```

We don't need to know how the function works! All that matters is that the function declaration and the comments tell the developer that this function will return the path between 'start' and 'goal', that the path will be in the form of a linked list of node indices, and that it describes what it expects as input in order to do its work.

Thereafter, if I need to use that function in my program all I need to do is:

- Set up an adjacency matrix for my graph (size NxN)
- Select the nodes I want to find a path between
- Call the function!

That was easy, wasn't it?

Except... wait a second! what about N? how is N specified? *we can't decide what N should be*

based on the API we provided! let's try again:

```
intList *findPath(int Adj[N][N], int N, int start, int goal);
```

Much better! now the user can let our path finding code what the size of the graph is. So that's all we need right?

Along comes another developer, and they have in mind finding a path between a 'start' location and any of a number of possible goal nodes (e.g. I want a path from my current location to ***any*** nearby pastry shop). Well, with our current API the developer is stuck doing this:

- Set up an array of possible goal locations
for each location *j* in the goals array
    path = findPath(A,N,s,goals[j])
    if path is not NULL break

Which is not too bad, but this seems like a common-enough ***use case*** that we may want to provide the API with a function to to this, so now we re-define our API to work like this:

```
intList *findPath(int Adj[N][N], int N, int start, int goals[k], int k);
```

And add to the documentation that *'goals[k] is an array with k possible goal nodes'* and let the user know *what the function will do when there are paths from the start node to many of the goals* (i.e. the developer needs to know how the returned path is selected among possible paths).

The questions you should have in your mind at this time is: ***is this better? is the above API more useful? is it easier to use and more general?***

As it happens, the answer may not be straightforward, and likely there won't be an answer that suits every developer who wants to use your module.

Example: You release your module on GIThub, and you get all kinds of happy comments from grateful users, except for one who lets you know their graph is too big and they can't keep an adjacency matrix around, can't you provide a function that works on adjacency list?

That sounds reasonable, but you can't really just expand the function call that uses the adjacency matrix to also work with an adjacency list ***(and this is a problem we will come back to again soon, we can solve it, but in C it would be pretty ugly, so we should solve it with a more advanced language)***.

For now, you decide to help this developer out by adding one more function to your API:

```
intList *findPath_l(intList* A[N], int N, int start, int goals[k], int
k);
```

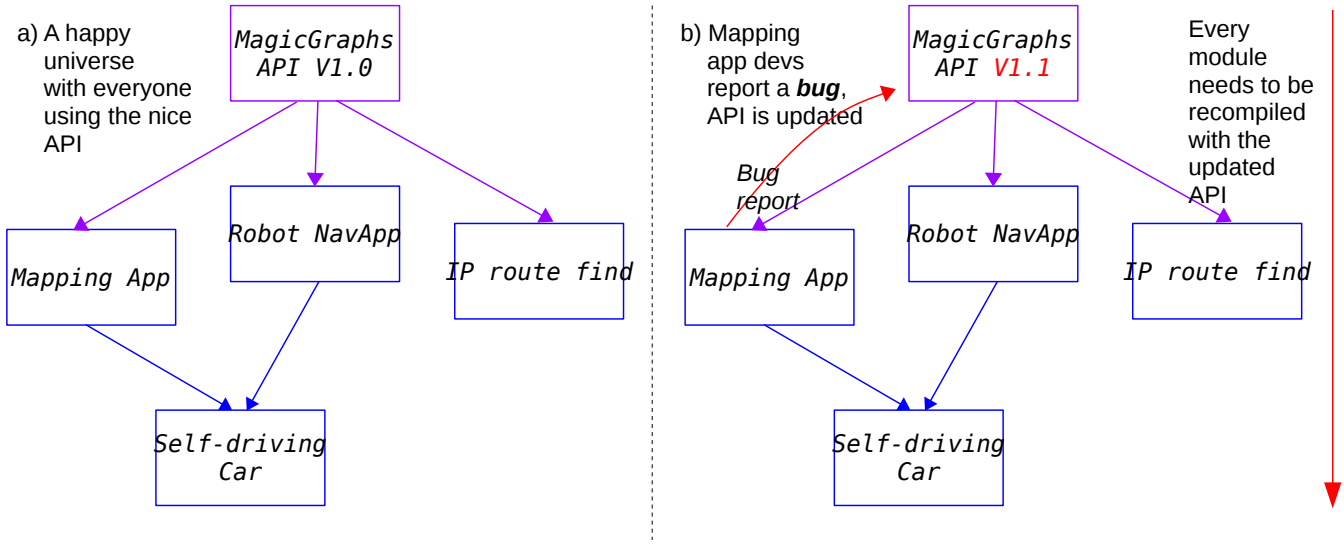
This function takes in an adjacency list in the form of an array of pointers to linked lists with the neighbours of each node. It will allow users to handle much larger graphs as long as the number of edges per node is reasonably small. ***So all is well with the universe right?***

What if - someone finds a bug in *'findPath()*'. The bug is with the actual path finding part of the function, and is independent of how the graph is stored.

- We can fix it
- But now we have two functions where the bug may be found, we have to fix both

So every time we add a variant of 'findPath()' to support a specific ***use case*** *we are increasing the amount of work that needs to be done when bug-fixing and maintaining the module*.

In general: You want to be very careful what functions you provide, and how they are declared. Making changes to the API once it's *out there* can become very problematic as shown in the diagram below.

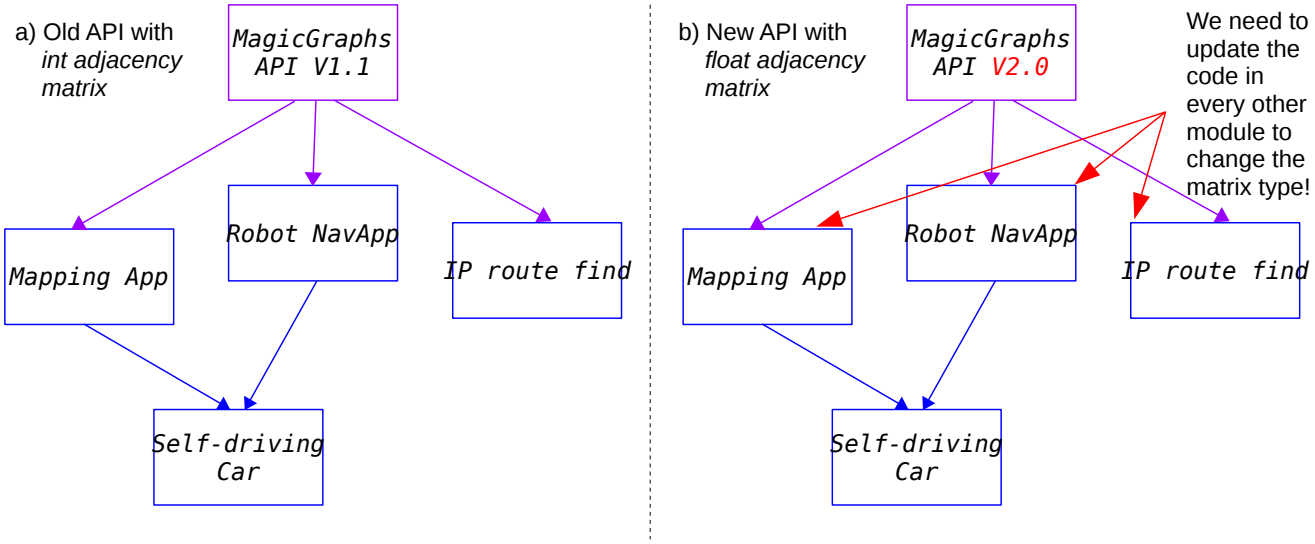


Each time that the API is updated, applications using it need to be re-compiled (this is especially important if your API has any security-sensitive components). ***So we need to think about how to organize our software so it gets updated whenever the APIs are updated***.

Moreover, once the API is reasonably stable, and is being used by a growing number of applications, it becomes very difficult to make serious changes to it ***even if we discover it wasn't designed well in the first place***. In the example above, suppose that now the '*RobotNav App*' requires that the *edges of the graph have weights* (e.g. to represent the *time* it takes to go from one place to another in the real world locations represented by graph nodes).

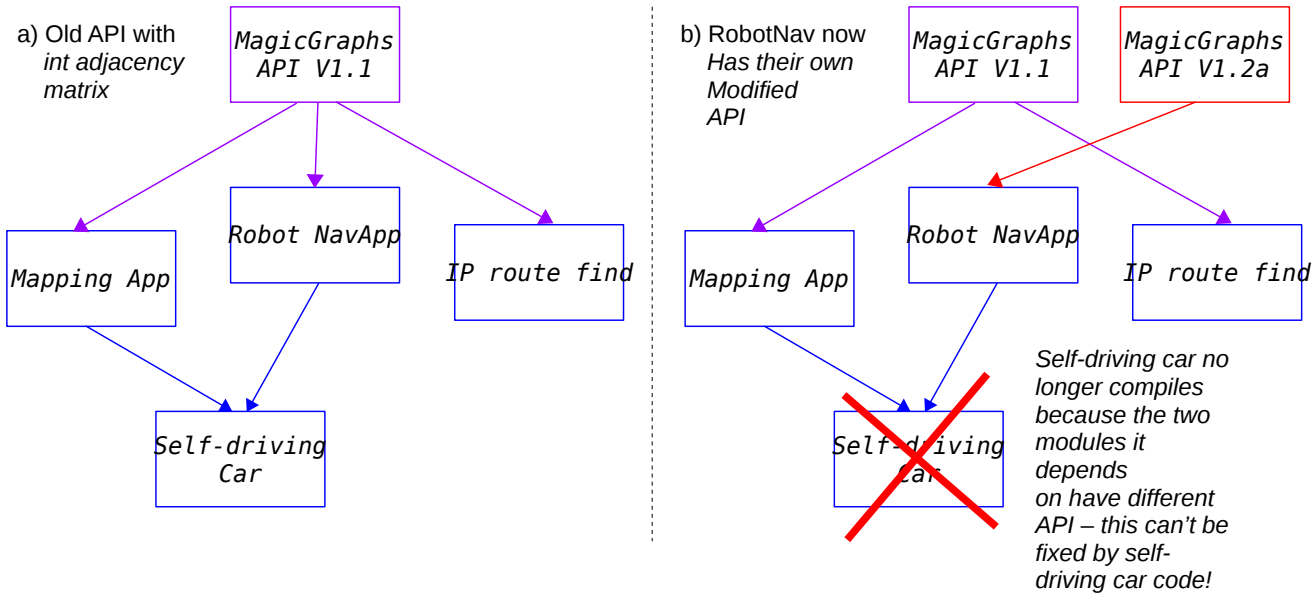
This looks like a *minor* change to the API right? Just update the adjacency matrix to be of type '*double*', or '*float*', and check that the path finding code works with real-valued edge weights.

However, you start thinking about what this means for everyone else:



Because you can not simply *cast* an array of *int* to an array of *double or float*, if you change the API to support what '*Robot Nav*' wants, every other application using your API will have to be updated (this now means changing the code, not just recompiling!) so that it uses *float or double* as the type for the adjacency matrix. This is not a nice thing to do to developers that were initially told to use *int*. ***The more your API is used, the harder it is to change fundamental aspects of it because of the impact to all the software that relies on the API***.

Once the API is stable and in use, it's a real problem if we didn't think of a ***use case*** that turns out to be important to many potential users out there. In the example above, suppose you get in touch with the developers of *'Robot Nav*' and tell them how to get into the code of the API and update it themselves to get the functionality they need. You get the situation below:



So as you can see, we have to be very thoughtful when we set out to design an API, and we have to think very hard about what ***use cases*** the module we are writing may be required for, and about how to develop the API in such a way that it will be reasonably easy to maintain and expand without a significant impact to applications that use it.

Question: How likely is it that the next version of '*stdio.h*' will change the definition for *printf()*? If it ever turns out we need a more ***crunchy*** print function, how do we introduce it into *'stdio.h*' without breaking thousands of pieces of software that already use this library?

Note: The discussion above introduces a couple very important concepts:

use-case - this is simply a *specific* situation or problem that a module or software component may be used for.

dependency - is a relation between software modules where the *dependent* module *requires* that a *library or module it depends on* be available and have the correct version in order to compile and run.

Anything you add to your code via '*#include*' statements introduces a dependency in your code. For instance, your programs most likely depend on the *system libraries 'stdio'*, and *'stdlib'* at the very least. If these are not present, you can't compile your program. If they have the wrong version, your executable may not run and needs to be re-compiled.

Note: You will learn a lot more about ***use cases***, ***user stories***, and the process of building software to fit what a ***user needs*** in the ***Software Engineering*** courses C01 and D01.

Advice from the experts on building a good API
(this is from Google's How to Design a Good API and Why it Matters:
https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/32713.pdf)

According to Google's experts, a good API should be:

- Easy to learn and use
- Difficult to use incorrectly
- Easy to maintain (the code in it is readable and well written)
- Easy to extend and improve
- Suitable for those who will be using it

The principles they propose for you to consider are:

- Your API should do one thing, and do it well - the functionality should be easy to explain and make sense of. If it can't be explained in simple terms, it's probably not well done.

- Your API should be as small as possible, but not smaller. It should satisfy the need it's serving, but do not try to add every single possible thing you can think of. The key thought here is that ***you can add to it later, but once it's in there, it's hard to remove functionality***.

- The API should be ***implementation independent***. This is a particularly important point. It should be possible for you to completely change the way a function is implemented, without needing to change the API. This also makes it possible to provide API implementations for different systems and platforms - they should be identical as far as the user is concerned.

- Maximize information hiding - Only make available to the user the functionality and data that the user needs. *This is harder to do with C, but consider that every function's data is local and can't be accessed outside the function.* We will see shortly how to do a much better job of controlling what data and functions a user of an API has access to.

- Names should be self-explanatory. Use naming consistently, the same goes for the use of underscores and capitalization.

- Good documentation is important. Every component must be documented properly.

- Think about performance, and avoid decisions in the API that will have a negative impact on performance.

- The user of the API should not be surprised by the behaviour of the API.

- The API should report errors as soon as possible after they occur (e.g. generating a chain of exceptions that is many functions long is not the best way to let anyone figure out what happened).

- If the API makes information available in strings, provide functions to parse the string into any components the user may need to handle separately. This prevents annoyance and keeps the user from having to write parsers for the API's output.

The items below are general principles of good programming, not just APIs

- Use the most appropriate return value for each function.

- Where functions have similar lists of parameters, be consistent with the ordering of the parameters (reduces the chance of the user making a mistake because they got used to the parameter ordering in a different function).

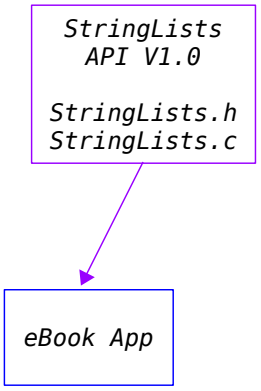
- Avoid long parameter lists

The Google document has several more recommendations specific to Java and object oriented code, so be sure to check back as you're going through your B07 course. For now, keep in mind the above, and note that even Google developers admit that designing and implementing a good API is a hard task, and that you can never achieve perfection. But you have to do a good job!

6.- Limitations of our programming language

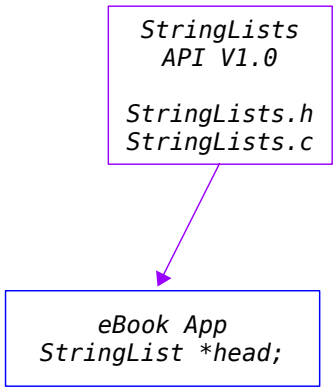
Once we start working with APIs, and thinking in terms of *modules* that are *self-contained* and provide functionality to the users without them needing to know the internal details of how this functionality is implemented, we will realize that there are limitations to what we can do with C as our programming language.

To illustrate the key issue that should concern us here, suppose we're providing a very nice module for creating and managing linked lists of strings (this could be used by an app, for instance, to keep the names of all the eBooks a user has in their cellphone). You have determined the appropriate API, and written both your (very well documented) header file, and the corresponding implementation file. Any user needing a linked list can include your module in their code and go. It looks like this:



So far so good. But now think about what we have learned of linked lists in C. *The pointer to*

the head of the linked list is kept and managed by the function(s) that need the list!



We haven't even started using the API, and we have already run into a problem: ***One of the key components of the linked list, the head pointer, is declared and kept by a function outside our API***. This is not great - the user of your module has access to all the data definitions for compound data types, and can declare and initialize as many as they want ***without calling your API at all***.

It's actually worse than that - because the applications using your API have direct and ***unrestricted*** access to the data in the linked list, ***they can go ahead and modify the list themselves without using the API.***

This may not seem like a big deal - after all, if they're using the API one assumes they intend to use it to save work and won't go around poking into the list themselves. *But*, there will always be users who will find an opportunity to do something your API doesn't provide by directly changing data in the list, and you also have to worry about *malicious users* who will try to mess with data in the linked list to see if they can get your API to crash, or to grant them access to privileged information they shouldn't have access to.

This breaks the model of a *module* being a *self-contained* entity that can be used without knowledge of the details of how the module is implemented. It introduces the potential for *bugs* created by users unintentionally modifying data needed by the module, and it introduces a security concern because of the potential for misuse by malicious users.

The root cause of the problem is that C ***provides no way to hide or protect sensitive data from being mis-used or accessed in a way that was not intended by us when we developed the module***. We can't solve this in C. So at this point we should look beyond the language we've been using and ***find a different way to organize our code and data*** so that we can implement software modules that are truly self-contained, and that are able to ***control access to the data managed by the module*** in a way that prevents users from unintentionally, or intentionally, mis-using the module and its functionality. This is called ***information hiding***, and is one of the fundamental principles of good software design.

The model of programming we will explore next is called ***Object Oriented Programming (OOP)***. It is one of the fundamental building blocks of software design and software engineering, and it

will allow you to build *software components* that are flexible, powerful, and self-contained.

7.- Object Oriented Programming

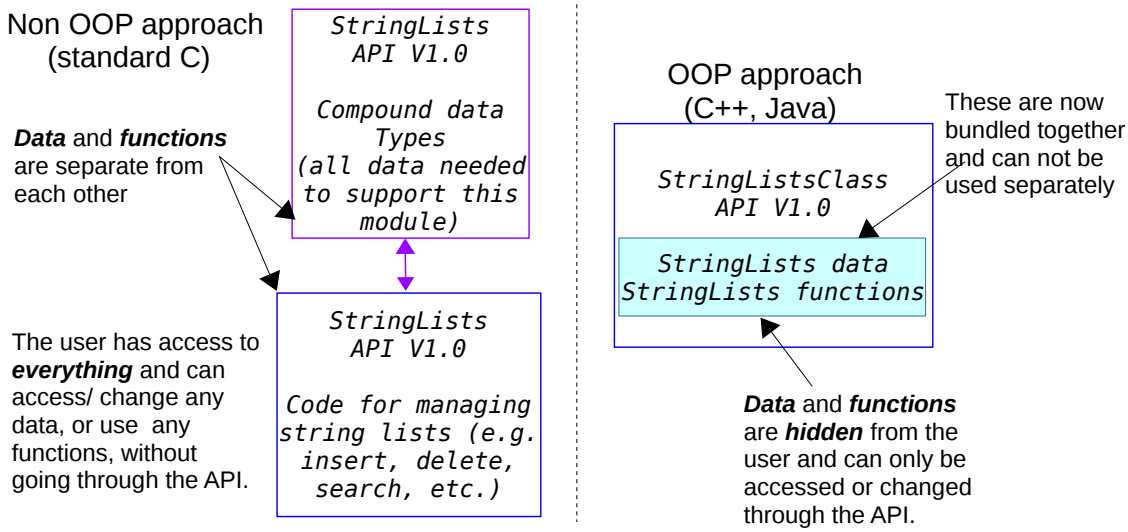
Object Oriented Programming is a model for developing software components that is based on the idea of ***encapsulation***.

Encapsulation means ***wrapping together*** all the components required to implement the functionality of a specific data type, data structure, or software module. This includes ***all the data as well as the functions that manipulate it***. It requires ***access control*** to data and functions that manipulate it in such a way that the designer of the module can determine what data and functions will be accessible to the user, and what legitimate ***use-cases*** will be supported. Very importantly, ***encapsulation*** requires that we be able to ***hide data and functionality*** from the user thus preventing accidental or intentional misuse.

To support all of these goals, Object Oriented Programming introduces the concept of an ***object*** as the fundamental unit of information storage, processing, and manipulation. You are already familiar with objects from having worked with them in Python during A08, now we will spend a bit of time understanding how they are built, what features they have, and what we can do with them as *software designers*. But in order to ***actually understand what an object is***, we first need to learn how to design and implement the principles of ***OOP*** that we discussed above.

Classes

In Object Oriented Programming, we expand the idea of compound data types (you already know very well what these are and how they work, but it wouldn't just to review Unit 3) so that we are not limited to storing data. With OOP, we want to bundle together all the data and all the functionality required to manipulate it. The resulting ***entity*** is called a ***class*.** This is illustrated in the figure below.



The idea of bundling together the code and data that comprise a single module, data type, or data structure, is important, but you have already worked with compound data types and you have already used objects in Python, so you know there really is nothing fundamentally ***new*** in the concept of a class. The importance of the ***class*** idea lies in how the ***programming language*** uses classes to provide you with ***information hiding*** as well as a number of other powerful features that were very difficult to implement without the support of ***object oriented languages***.

In OOP, the data components of the class are called ***member variables***, and the functions that implement the functionality for the class are called ***class methods***.

Information Hiding in Classes

Recall that in C all our variables and functions have an associated ***data type*** that is fixed and ***used by the compiler to determine what code needs to be generated*** to implement the functionality specified in your program.

In ***Object Oriented Programming***, both the ***member variables*** and the ***class methods*** have an associated ***access modifiers***. These specify the ***visibility*** of each of these components much the same way that ***scope*** determines the visibility of variables within the code.

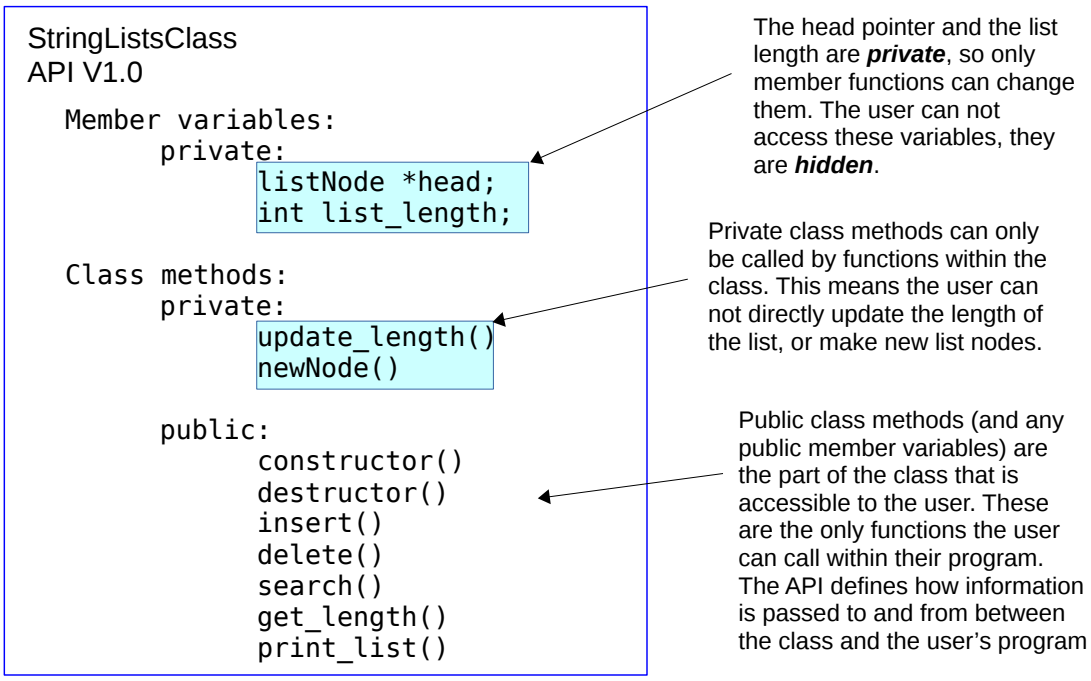
The smallest subset of ***access control modifiers*** that has to be implemented by an object-oriented language is:

public - This modifier states that a ***member variable*** or ***class method*** can be access by any code ***outside the class*** that contains it. It is appropriate for all the components of the API that the user needs in order to gain the functionality provided by the ***class***.

private - This modifier states that a ***member variable*** or ***class method*** can be accessed only by ***class methods*** belonging to the class itself. The user and any code written outside the class can not see, access, or use these ***protected*** members of the ***class***.

There are further modifiers that may or may not be available depending on the language you are using, but the two above constitute the minimum subset that will allow you to implement ***information hiding*** and ***encapsulation***. The diagram below shows how a class implementing our string list API could be structured – it shows both the member variables, and the class methods. Access modifiers indicate which components of the class are visible to (can be accessed/used by) the user's program. Any private members are only accessible from within the class.

For example, the '*head*' pointer is not accessible to the user, it can not be changed from outside the class. However, functions such as '*insert()*' which are ***class members*** can access and change the value of the '*head*' pointer. In this way, ***we expose to the user only the functionality of the class that we want to provide,*** and can do so in a way that prevents misuse of the class and its member variables and methods.



There are ***two methods we haven't seen before*** in the class model above, the ***'constructor'*** and the ***'destructor***'. These two functions have an important role the class:

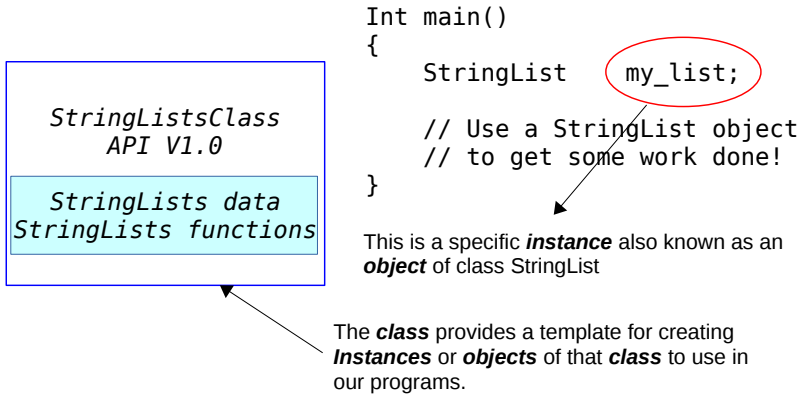
- The ***constructor*** – is ***automatically*** called by the compiler when we declare a new ***instance*** of the class (e.g. when the user declares a *StringList* in their code in order to use it for managing their list of strings). The ***constructor*** has the job of ***initializing*** the member variables and any data that the class will need to suitable values. In the example above, we could expect it to set the '*head*' pointer to ***NULL***, and the '*list_length*' to zero. For more complex classes, the ***constructor*** may do a lot more work.

- The ***destructor*** – is called when an ***instance*** of the ***class*** goes out of scope or when we want to delete an ***instance*** of the ***class***. It has the job of *cleaning up* after the class. For instance, in our example above, the ***destructor*** would be in charge or freeing all memory allocated to the nodes of the linked list. For more complex classes, the ***destructor*** may do a lot more work.

The important thing to keep in mind is that these methods are provided in order to ***automate*** what you have been doing all along up to this point: *creating and initializing compound data types*. Remember that so far we have always needed to write a function whose only task is to do that. With Object Oriented Programming, this is built into the class. As we will see later, combined with other features of OOP, this gives us quite a lot of flexibility and power in terms of how a new ***instance*** of a class is initialized.

Wait a second: *just* ***what is an instance?*** It's a term used in ***OOP*** to indicate a particular *variable* of a given class – but since these are much more crunchy than, say, an *int* or a *float*, we call them ***instances*** or ***objects!*** - And this explains the term ***Object-Oriented Programming***.

- The ***class*** is the template for building ***objects*** that have all the member variables and class methods from the ***class*** and can be used within a program. In terms of what you learned with C, the ***class*** is the equivalent of a ***typedef***, and an ***instance*** or ***object*** of that class is analogous to declaring a ***variable*** of a particular compound data type.



8.- Building a class in C++

Now that we're talking about Object-Oriented-Programming, we need a language that can support the features we need to build and manipulate classes and the objects we create from them. C is not designed for that, and will impose limitations on what we can do that will make it very hard for us to any reasonable job of implementing and working with classes and objects.

Therefore, let's have a quick look at C's more powerful and more flexible descendant: C++.

Please note: We don't expect you to learn C++ from scratch in a couple weeks! We will only take a few steps into the world of C++ so that you leave A48 with ***a concrete idea of how object-oriented programming works in practice***. But you're not expected to learn to implement complicated programs in C++. That would require its own course! Instead – You want to pay attention at how the principles of ***OOP*** are exemplified by C++ class declarations, to how the class' methods work, and what kinds of things we can do with objects once we have created them. You will have plenty of time to develop programming skills in an object oriented language next term, with Java in B07.

Most of what we will do here should be familiar to you, since you worked with objects in Python during your A08 course.

C++ was developed in the 80's as an extension to C (hence the name!), it was provided with the syntax required to support object-oriented programming, and has been continuously expanded and improved over the years. C++ is one of the more important languages in serious software development where speed is important. Areas such as operating systems, security, networking, embedded systems,

media-related software (e.g. encoding/decoding video and audio), compilers, and many others are important users of software written in C++.

You should keep in mind that C is a subset of C++, so all your C code, everything you have learned so far, can and should be used when you develop in C++. In effect, C++ is just giving you additional superpowers for dealing with objects, but you can still do everything else you've been doing up to now with C.

With that in mind, let's see how a class is declared in C++. While you think about it, consider in what ways it is similar, and in what ways it is different from a ***typedef*** from C.

```
typedef struct ListNodeStruct
{
     char string[1024];
     struct LastNodeStrucy *next;
} ListNode;

class StringList
{
     // Member variables
     private:
          ListNode *head;
          int list_length;

     // Class methods
     public:
               StringList()
               {
                 head=NULL;
                 list_length==0;
               }
               ~StringList();
          void insert_string(char string[1024]);
          void delete_string(char string[1024]);
          ListNode search(char string[1024]);
          void clear_list();
          int  get_length();
     private:
          void set_length(int l);

};
```

A few things to note in the example above:

- The ***typedef*** for the ***ListNode*** is ***not part of the class***, but we need it to define the nodes of the string linked list we're building.

- The ***class declaration*** is just like any compound data type declaration you've done before in C, except we say '*class*' so the C++ compiler knows we're bundling data and functions into one nice package.
- All the ***member variables*** in our class are ***private***. No functions or code outside of the class can access or change these variables. They are ***hidden*** from user code.
- The function ***'`StringList()`'*** which is the class ***constructor,*** has no return value or type. It gets called automatically when we create ***objects*** of this class, and will initialize the class' member variables as needed. In the case of our list, it initializes the head pointer to ***NULL***, and the '*list_length*' to zero. This function has the same role as the *'__init__'* function you have used before with Python objects.
- The function ***'`~StringList()`'*** is the class ***destructor.*** Like the constructor, it has no return value or type, and gets called automatically when an ***object*** of the class goes out of scope or gets deleted. Its job is to ***clean up*** after the class – so in effect, free any memory that was dynamically allocated by class methods, close any open files, etc.
- The remaining ***public*** methods are analogous to the functions you would normally find in any regular linked list. We have *'insert_string()*', *'delete_string()'*, and '*search()'*, the only interesting thing about them is that the *'search*' function ***does not return a pointer to a node in the list!***. In ***C***, we used to return a pointer to the node that contains data for the user query. But in OOP this would break the principle of ***encapsulation!*** we ***do not want the user to get pointers to the class' internal data***. So instead, the search function ***returns a copy of the data stored in the node that corresponds to the user's query***. This way, the user gets the data they need, but does not have access to the list nodes themselves, and therefore can not change their content without going through our class' API.
- There is one ***private*** class method: '*update_length()'*. This method can only be called by other methods within the class. It can not be directly called by the user.

Please keep in mind that the above is just a very small example so you can get a concrete idea of how C++ defines a class, and what the structure of the class looks like. There is a lot of refinement you can add to classes in C++. We show you what a class definition looks like in C++ so you can have a ***concrete picture in your mind*** about how the ***general principles of what classes are and how they are organized*** are implemented in one of the most common object oriented languages. You will have plenty of time to learn a lot more about how to declare and use classes during your B07 course, with Java.

Method Overloading

Recall that when we were studying APIs, we ran into a situation where one of the users of our graph management library wanted to use an adjacency list instead of an adjacency matrix. With C, we found ourselves stuck either changing the API in a way that would ***require all modules using our API to be modified and recompiled***, or ***have the users who want a different API call to use their own custom version,*** with the downside that their code would then not be compatible with any other module using the standard API.

In C++, we have the flexibility to declare ***multiple versions*** of a function or class method, each with a different set of input parameters. This solves precisely the problem we had in C where once the API is defined it's very difficult to change the function declarations.

Suppose that instead of having written our API in C, we had used C++. We would have defined a ***class*** to store and manipulate graphs, and within that class we would have a public method:

```
intList *findPath(int Adj[N][N], int N, int start, int goals[k], int k);
```

Which is identical to the API call we had defined in C. When we find out that a user wants to be able to use an adjacency list, we can simply add an ***overloaded function declaration***, with the same name, but with the parameter types our user needs (and even with a different number of parameters!). So now our ***graph management class*** would have the following class methods:

```
intList *findPath(int Adj[N][N], int N, int start, int goals[k], int k);
intList *findPath(intList* A[N], int N, int start, int goals[k], int k);
```

Now the class has two methods with the same name - the compiler will select which one to call based on the user's code (by looking at the types of the parameters the user is calling the *'findPath()*' function with). Suppose that we have another user now who wants to use *floating point* adjacency matrices, not *int*. Well, we can expand our API by providing an ***overloaded function*** that accepts a floating point adjacency matrix:

```
intList *findPath(int Adj[N][N], int N, int start, int goals[k], int k);
intList *findPath(double Adj[N][N], int N, int start, int goals[k], int k);
intList *findPath(intList* A[N], int N, int start, int goals[k], int k);
```

We can have as many different versions of the function as we may need, ***but we should be careful*** that each of these supports a ***valid and distinct use case***. Recall that our API should be as small as possible! so providing every possible combination of input parameters that any user may need is ***not good design***.

Note: This is true for all your coding, but particularly important for function overloads: Make sure that similar parameters are declared in the same order. This will reduce the chance of mistakes by the user of your module, and will also enable them to learn a pattern to how you implemented things inside your classes.

What you should understand up to this point:

- Limitations of non-object-oriented programming
- Encapsulation and ***information hiding***
- What is a ***class***, and what are the ***components of a class***
- Access modifiers for ***information hiding***
- ***What is an object*** (difference between an ***object*** and a ***class***)

9.- Polymorphism and Inheritance

There are two more fundamental component of object-oriented programming that we need to study before sending you on to your Software Design course next term: ***Polymorphism and imheritance***.

We call ***inheritance*** the ability of an object-oriented language to build and use a ***hierarchy of classes*** representing ***entities that are related to each other and share some characteristics*.** This turns out to be a very common situation that is hard to deal with in non-object oriented languages like C.

To provide a concrete example, consider the problem of implementing a music synthesizer in your computer - the synthesizer takes in a linked list of musical notes, ordered by their position in a musical score. Its job is to play these notes by generating the sound corresponding to each. ***The tricky part is - it needs to be able to play multiple notes at the same time, and these may belong to different musical instruments.***

In a non-object oriented language, we have a little bit of a problem:

- Musical notes from different instruments have to be handled in different ways. For example, for a guitar sound, the sound generating code may have to simulate the guitar string with a suitable algorithm, whereas for a piano sound, it may just play a pre-recorded piano sound sample. The details don't matter, what matters is ***each instrument's sound is generated in a different way***.
- However, other properties of a musical note are ***shared***. Each note will have a ***frequency***, and a ***duration***, and a ***position in the musical score***, plus any other information needed by the synthesizer to organize and manage the notes it's playing.

Let's see what happens when we try to implement ***just two*** different note types in C. We know we need a compound data type to represent the note data. ***We need to make sure there is a single data type*** that can represent ***any*** note, because otherwise we can't put all the notes into a single list to be played by the synthesizers (can't mix and match pointers to different data types!).

So we start with the things that are ***shared*** by every note from every instrument:

```
typedef struct note_struct
{
     double time_position;      // Note's position in the musical score
     double frequency;          // The note's frequency
     double duration;           // How long the note is played
     int        volume;         // The loudness of the note

     double *note_data;         // Pointer to the note's data array
} note;
```

So far so good. Right?

Somewhere in the synthesizer, we will have code that takes a note, and generates the sound sample corresponding to the note's sound at a given point in time.

```
double get_sound_sample(note *my_note, double  time_index)
{
     // Computes and returns the sample of sound to be played for this
     // note at the specified time index.
}
```

Nothing special so far. But here's where we run into trouble: ***How does the*** '*get_next_sample()*' ***function know what type of instrument this note is from, and how to play it?***

We don't have a lot of good options here, given what C can do for us. A simple solution is to add another variable to our '***note***' data type:

```
typedef struct note_struct
{
     double time_position;      // Note's position in the musical score
     double frequency;          // The note's frequency
     double duration;           // How long the note is played
     int    volume;             // The loudness of the note
     int    instrument_ID       // Indicates which instrument this note
                                // belongs to.
     double *note_data;         // Pointer to the note's data array
} note;
```

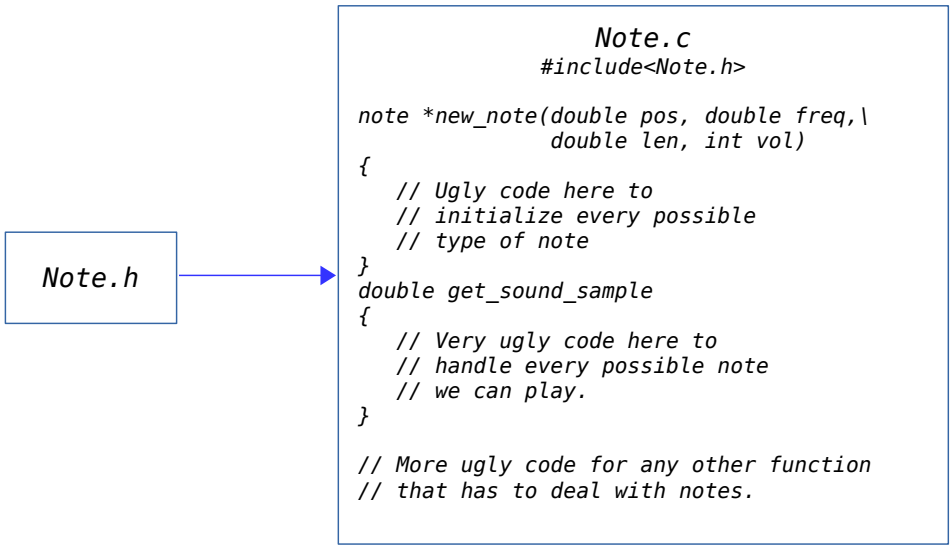
Simple! we assign to each note the *ID* of the instrument it belongs to, and we're done, right? except now we have to have, in ***every function that handles notes***, code that looks like this:

```
double get_sound_sample(note *my_note, double  time_index)
{
     // Computes and returns the sample of sound to be played for this
     // note at the specified time index.
     if (note->instrument_ID==0)
     {
          // Do what corresponds to instrument ID 0
     }
     else if (note->instrument_ID==1)
     {
          // Do what corresponds to instrument ID 1
     }
     else if ...

     // Or you can use a switch statement, but in any case you have one
     // different case for each instrument, and the code is unnecessarily
```

```
    // long, difficult to test and debug, and difficult to read!
}
```

If you still think the above is not that bad, consider that the typical synthesizer supports ***hundreds of different instruments***. You can imagine the code for *'get_sound_sample()'* is going to be ugly. The same will happen to other functions that handle ***notes***. For example, the function that allocates and initializes a new note will likely need to do different things for different instrument types (for example, initializing the data array for the note, which will happen in different ways for each instrument). An overview of how our implementation of a ***note*** module for the synthesizer looks like this:



Not a fantastic solution, if we consider all we know about how to write good software! On top of the code being less than elegant, there is ***no information hiding***. For example, nothing prevents a user's program from ***changing the note type while the note is playing***, which may sound interesting but is definitely not intended behaviour, and could easily create bugs or worse.

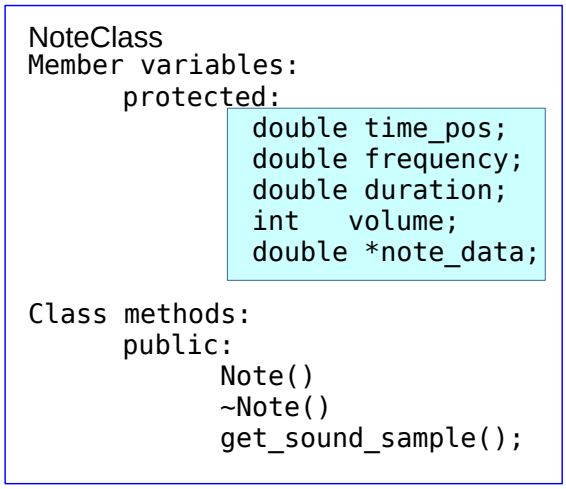
The problem we have stems from ***trying to implement a set of closely-related, but not identical data items using a single compound data type***. With object-oriented programming and through the use of ***classes***, we can find a better way to model, represent, and implement such data items.

Here are the properties we want from our model

- We want to ***maximize code reuse*** - Any code that is shared among our ***classes*** should be implemented only once.
- We want to ***minimize data duplication*** - Common variables and data should be declared only once.
- We want to be able to ***extend*** and/or ***refine*** the behavior of ***any of our classes without affecting the rest***.
- We want the model be able to show how our ***classes are organized, and how they relate to***

each other.

This leads to a ***hierarchical representation*** of related classes - where common data and methods are implemented by a ***parent class***, and ***specific behaviour is implemented by derived (children) classes*** - let's see how this would look for the synthesizer example above.



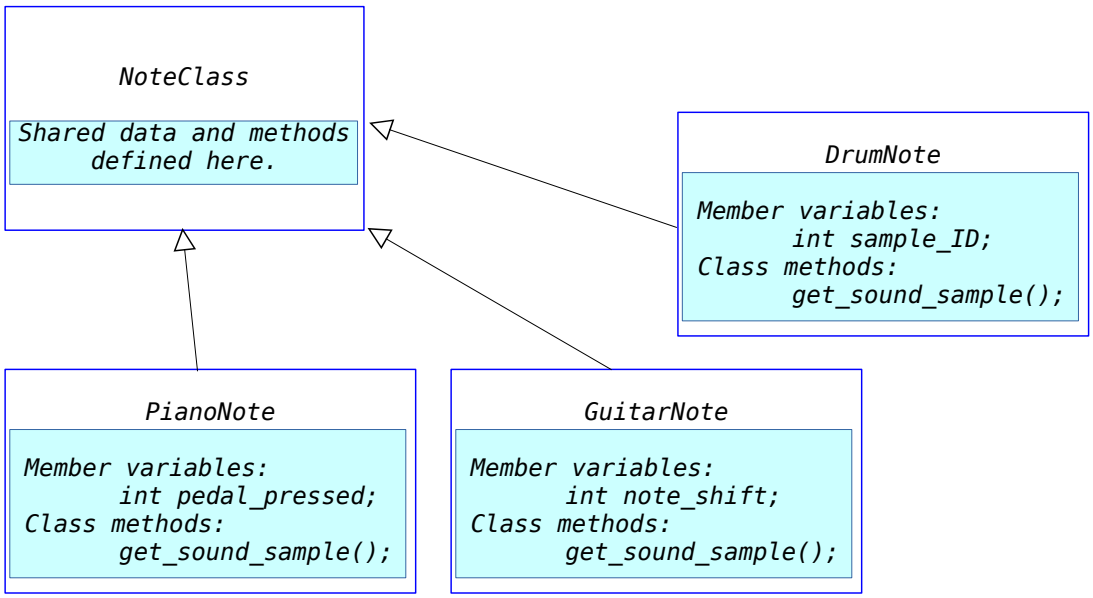
The *'NoteClass'* above defines the ***shared characteristics*** that ***all notes our synthesizer plays must have***. This includes the common member variables that every note has, and fundamental methods needed to handle a note, including a ***constructor, destructor***, and the function that ***returns a sound sample for the note***.

We have a new ***access modifier: 'protected'***, the behaviour of this access modifier will become clear in a moment.

With object-oriented languages, we can take the *'NoteClass'* and create any number of ***derived classes*** (also called ***subclasses***), all of which will ***inherit*** the ***public and protected*** member variables and class methods from the ***parent class*** (also called ***the base class***). Each ***derived class*** can then go on to add their own member variables and methods, ***and can also provide implementations of the shared methods defined by the parent class, but refined to implement behaviour specific to the subclass.***

The '***protected***' access modifier is used to give ***derived classes*** access to a ***parent class***' data that ***still must be hidden from user code***. Think of it as an extension of the '***private***' access modifiers, where access is now given to ***derived classes*** while preserving ***information hiding***. Any member variables or methods declared '***private***' by the parent class are still only accessible within that class!

An illustration of how our class structure would look for the synthesizer notes is shown below.



In the diagram above, we have ***three subclasses***: *'PianoNote'*, *'GuitarNote'*, and *'DrumNote'*. Each of them ***inherits the protected and public components of*** *'NoteClass'*. So, they all have *'frequency'*, *'time_pos'*, *'duration'*, *'volume'*, and *'note_data'*. They also ***inherit*** the public method *'get_sound_sample()'*.

However, each ***subclass*** has gone on to declare additional member variables suitable to what that ***specific type of note*** needs. And very importantly ***they provide their own implementation of*** *'get_sound_sample()*', so now each ***derived class*** is responsible ***only for the implementation of code specific to that subclass***. No more ugly code full of conditionals!

What have we gained?

- There is no un-necessary data or code replication: Any common member variables, and any shared methods that are identical for every class in our hierarchy are declared and implemented only once, in the ***base class***.
- We can have any number of ***subclasses***, each of which can declare their own member variables specific to the subclass.
- ***Subclasses*** can ***re-define*** common methods declared by the ***base class***, and in this way we confine the code that is specific to each ***derived class*** to that specific ***subclass***' implementation.
- Finally, and most importantly: ***Subclasses can be used with any code that handles the base class***. In simple terms: for our synthesizer application, the user need only implement code to create and handle a list of ***'NoteClass***' objects, and in this list we can insert objects from any of the ***subclasses*** (we can have piano notes, guitar notes, and drum notes together!). They are, after all, musical ***notes*** - all of them!

That last item explains the term ***polymorphism***: We have a class of objects (musical notes) that can have many different *shapes* - yet they can all be used as musical notes.

How this would look in practice (pseudocode):

Play a few notes:
*Initialize a linked list to contain objects of **NoteClass***
*Insert notes to be played into the list (can be of **any subclass**).*
- Insert a few PianoNotes
- Insert a few GuitarNotes
- Insert a few DrumNotes
To play the notes:
*Traverse the linked list, and for each **'note'** in the list call*
note->get_sound_sample()

The linked list knows ***nothing*** about the specifics of each ***subtype of note***. It deals only with pointers to objects of ***'NoteClass'***. But the language knows the ***subtype*** for each note in the list, so when the *'get_sound_sample()'* function is called, ***the implementation from the right subtype is called automatically.***

We do not need to get into the implementation details here - they are fairly involved, and require plenty of time to learn well - there will be time for that later. What is important is that you understand these ideas:

- ***Inheritance*** allows us to design and implement a ***hierarchy*** or classes so that we can have variations (as many as we want) of a ***base class***, each with their own data and methods, each able to implement behaviour specific to the ***subclass***, but at the same ***the user's code only needs to worry about handling objects of the base class.***
- ***Subclasses*** can re-define ***methods*** from the ***base class*** to implement subclass-specific behaviour.
- We have a way to preserve ***information hiding*** while allowing ***subclasses to access variables and methods from the base class that are hidden from user code***. This is thanks to the ***'protected'*** access modifier.

Designer beware:

You can create a class hierarchy as complicated as you like - but if you don't think about it carefully while designing your software it's easy to end up with a mess that is much worse than what you would have if you had stuck to standard C (with all its limitations).

Therefore, to seriously learn how to design and implement class hierarchies that have been designed to solve specific ***use cases***, you need to learn and understand ***design patterns***. A good part of your B07 course will be devoted to exploring and understanding these patterns so you know how to implement them, what use case they are designed to solve, and why they represent the optimal organization for classes that solves a particular problem.

If you can't wait to get started learning more about software engineering, I'd suggest having a

look at a well known and extensively used set of principles that guide the design of good software. These are known as ***SOLID***, and you'll definitely find it useful to be familiar with what they involve:

https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design
https://deviq.com/solid/

Abstract classes

One final idea worth mentioning: It is possible (and for some problems it is the appropriate solution) to define a ***base class*** that ***does not implement any of the shared methods***. The ***base class*** only declares ***what methods must be provided by subclasses***, and ***every subclass must provide an implementation***.

Such ***base classes*** are called ***abstract classes*** - they specify the ***shape*** of an ***object*** but none of the implementation details. You'll learn plenty about ***abstract*** classes in your B07 course - ***Java interfaces*** work in a ***similar*** way and provide powerful tools for building good software.

As an example in the context of our small synthesizer module, you could define an ***abstract class*** for an ***instrument*** – this class represents any specific type of musical instrument, but there is no implementation that can be attached to it. Instead, we would define in this class the shared member variables every instrument must have, and the ***methods that each derived class must implement***. We can thereafter implement specific instrument classes that ***inherit from the abstract instrument class*** and ***provide the implementations for all the methods required to get that specificd instrument to work.***

10.- A few notes to remember

The above is just avery quick introduction to the ***principles and ideas*** in good software design that you will expand upon in your software engineering courses. You should review them before you begin B07, and place every concept and idea you will learn there in the context provided by having experienced a non-object-oriented language, and having seen the limitations it has when it comes to designing and implementing complex software for general use.

At the same time, you should remember that object-oriented programming is just one of the models for building software, and is not expected to be the best possible model for every application or every problem. You need to learn to determine when object-oriented design is called for, and when you're likely to be better off without it.

For example, object-oriented code does have non-negligible overhead compared to straightforward non-oop code. In performance sensitive applications, you may want to consider the relative value of flexibility in software design, and the importance of information hiding, against performance penalties caused by object-based code.

You will learn more about other programming models as you work your way through your

program. Don't forget what you learned here!

11.- Wrapping it up!

This is also the end of our A48 course. It's been a long road, so you should take a moment to reflect on all the ground we have covered, and all the fundamental ideas you should carry with you as you go on to bigger things. You should re-visit your notes in the future, ***in preparation for covering related material in higher-level courses***. As a quick guide:

For Software Design - CSC B07: Unit 3, Unit 5, Unit 6
For Software Tools and System Programming - CSC B09: Unit 1, Unit 2
For Introduction to the Theory of Computation - CSC B36: Unit 4
For Design and Analysis of Data Structures and Algorithms - CSC B63: Unit 3, Unit 4, Unit 5
For Computer Organization - CSC B58: Unit 2, Unit 3

Some of the ***fundamental ideas*** that you should take with you from this course:

- ***Abstract Data Types*** and ***Data Structures*** - What they are, how they are helpful in thinking about the best way for storing, organizing, and manipulating large collections of data.
- ***Multiple common data structures*** - Arrays, linked lists, trees, and their variations. What are their properties, their advantages and disadvantages, and what kind of data-storage and manipulation they are best suited for.
- ***Computational complexity*** - As a means of comparing in an *implementation independent* way different algorithms, different data structures, and also as a way for understanding *how hard particular problems are*.
- ***Tree structures*** and how we used them to represent information, as well as how they provide *efficient* access to large data collections.
- ***Graphs*** and their properties. You should remember how to represent them, and how to use them to solve problems like path-finding, as well as for finding answers to interesting questions you can ask on connected data items.
- ***Recursion*** as a problem solving tool. What the components of recursion are, and how to form a recursive solution to a problem. You should also remember the kinds of problems that are naturally well suited for recursive solutions - including graph-based data structures like trees and nested lists, and hierarchical structures whether they are in computer graphics or in operating systems.
- ***Designing good software***. Many of you will spend some part of your professional life developing software. It is important that you do this well and that you spend time learning and practicing the skill of carrying out good software design.
- And of course: ***Programming in C*** - which will be useful to you even if you never program in C again, because it will have helped you develop the habit of ***thinking carefully before you code,*** considering the ***most appropriate data type for your information***, ***understanding what each line of your code is doing***, and ***thinking in terms of what happens in memory when you manipulate data***.

We hope the course has been useful for you also in terms of acquiring fundamental concepts

that will be needed later on, but also in terms of exercising essential habits and skills:

- ***Problem solving*** (all of those exercises and your assignments!).
- ***Study habits*** (preparation in advance of the lecture).
- ***Work habits*** (starting early and working consistently).
- ***Testing*** software (for correctness, not for user-acceptance).
- Seeing ***how different concepts relate*** to each other (not learning isolated facts/ideas).
- Asking ***why*** until things make sense.
- Knowing that you have ***really learned something***, by being able to explain it to someone else.
- ***Building a strong and healthy community*** for yourself and your colleagues.
- ***Being aware and thinking about social and ethical issues*** you will face through your career.

Don't lose the skills and habits you've worked so hard to acquire! practice them often, and keep strenghtening them - they will serve you well through your professional career!

We wish you success with your career going forward, and we thank you for all the hard work you've done this term without which all the work we poured into making this course really solid would be worth nothing.

So, good luck, and stay in touch!

(Your friendly neighbourhood course instructors: Paco and Marzieh)