Chapter 6 Designing and Building Good Software

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 are writing in a way that makes our solution as **useful to others** as possible.

In this Chapter, we will discuss the general principles that guide the **organization of larger pieces of software**. The goal is to understand **how and why** we need to think very carefully about the way we design the software we are writing, and to learn about the principles that have been developed to help us better organize, maintain, and expand on a given software solution.

The principles we will learn in this Chapter will be the foundation on which we will build a full understanding of the software design process, a topic you can explore in depth by reading up on software design, software engineering, and software architecture.

6.1 Software as a collection of modules

As we have worked our way through previous chapters, we have already been taking advantage of the significant amount of work done by others on our behalf. All the functions in standard **C** libraries that we have used to implement the different algorithms discussed in the book had to be **designed, developed**, and **tested** well before they shipped with your compiler so you could easily have things like **printf()** or **qsort()** at your disposal.

Question: what would software look like if we didn't have any libraries for commonly used functions? If we think about it for a moment, we will realize that such software would look like the illustration in Fig. 6.1.

Figure 6.1: Software as a disorganized collection of **stuff** that does **something**. **Courtesy of Randall Munroe, http://www.xkcd.com**.

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 unrelated 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 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). As an additional but very important benefit - these modules can be **easily reused**, which means that they can be either directly combined with entirely different applications we are developing, or re-used with only minimal changes.

In effect, what we want is to build a **large library of modules** that we can **reuse with ease** to build any software we want. For example - we may want to write a module that provides support for creating, storing, managing, and running standard algorithms on **graphs**. It could, for instance, provide path-finding using **DFS**, allow us to modify the **graph** by adding or removing **edges** and **nodes**, and handle all the required data structures so that we don't have to re-write all of that code whenever we need to use a **graph**. We want to write this **module** in such a way that anyone else writing a program that needs to do path finding can easily take our module for handling **graphs** and use it without **needing to read through** our code in detail, and without needing to worry about **our code being full of bugs** or **being unnecessarily inefficient** (in terms of complexity).

This is not a trivial problem - out software needs to be **designed carefully and correctly**, so that it can work on a **graph** representing any data a developer may need to handle, so that it can easily be called from software not written by us, and so that the resulting programs can be **tested, debugged**, and possibly **expanded** by other developers who don't know (and don't need to know) the details of how we developed the **graph module** or even how the algorithms contained in the module work.

Note

If you think back to Chapter 3, we already have discussed this idea. Back then we said that **ADTs** are useful because they provide a developer with a concise idea of how the particular **ADT organizes data** and what **operations** it supports in a way that is **independent from implementation details**. This allows a developer to use **any implementation of the ADT** in their programs at the level of **data and operations on data** without needing to know how these are actually implemented.

With well designed **software modules** we are pursuing a similar result: We want to design **modules** that can be used in terms of **data** being stored and manipulated, and a set of **operations and algorithms** that can be applied to the data managed by the **module**. We want a developer to be able to use the **module** without needing to know or understand how the actual **operations and algorithms** are implemented, as this will free them to focus on their actual task and allow them to build sophisticated software with advanced functionality quickly, reliably, and without being experts on every possible aspect of computer science that may be involved in solving a particular problem.

Let us look at some of the **key principles** that should be in our mind when we start exploring the world of software design.

6.2 A wish list for building good software

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 it 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.

1) Modularity: Our software is composed of separate **modules** each of which has **one specific task**, and each of which is **self-contained**, so it can be **understood, tested**, and **maintained** independently of the others. A well designed modular program will help:

- Reduce **replication** of code
- Improve the chance that code we write will be **reused**
- Make it **easier to test** the code and verify it is correct
- Help a developer focus on **the big picture** of how a particular software is structured and how it works

2) 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 the same functionality . For instance, if we develop a **module** that implements shortest-path finding between two nodes in a graph (which, as we saw, is a common problem in many application domains). We want our implementation to be such that anyone needing to find the shortest path between graph nodes can simply take our code and use it to build their application.

3) Extendibility: We want our software to be **easy to extend and improve**. This allows us to build better, more capable software over time by improving and expanding its functionality.

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

5) 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**. Our code should have been **reviewed** by experienced developers not related to its implementation, and a suitable process must be in place for **documenting**, **tracking**, and **resolving bugs** found after the software is released.

6) 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.

7) Openness: When possible (e.g. when we're not developing software for a company that has a claim of ownership over the code they pay us to write), we should consider contributing our work to the **open source software (OSS)** community. There is a lot of good work already out there that is 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. We will get back to this at the end of the Chapter, in the last section on how to build software that works.

8) Privacy and security: A significant portion, or even the majority of the software we write will be handling potentially sensitive information. The data the program works with should be **protected** from **unauthorized access, use, or distribution**. Consideration must be carefully given to the **privacy of a user's personal information**, and if the software is **accessible over a network** then we have to follow best known practices for **securing access** to the software, **protecting information from falling in the hands of hackers**, and **ensuring no unauthorized copies** of any information managed by the application can be made or distributed without proper authorization. Very importantly **the user must always grant informed consent** before any personal information about them can be requested and stored by our application. This means **clearly and transparently** indicating **what information will be requested and stored**, **how the information will be used**, **how long it will be stored**, and **how (if at all) it will be shared** with any third parties. **No information should be gathered without informed consent**.

The above is not a comprehensive list. Specific applications will require additional consideration to be given to factors such as **reliability**, **failure-tolerance**, **computational performance**, **data storage requirements**, and many other possible factors. However, the properties listed above are a solid starting point on which we can build good software and apply to the vast majority of applications we may need to develop.

6.3 How modules are organized and used in C

So far, we have been working with applications that contain only a couple program files at the most, and we have not needed to break up our application into multiple independent **modules**. This is only suitable for small programs, and for applications with 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, without any of the code. **Header files** also provide definitions for common things such as for instance **mathematical constants**. You have 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 such as **printf()** or **malloc()** that are provided by these libraries.
- **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. To create a working **executable program** we eventually need to have access to the implementation of **every function** that is used by the application, regardless of whether it is part of a library, or whether it is part of the code we have written ourselves.

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: this means bringing together the **executable code** that implements the functionality from each of the modules being used, and **updating the place-holders with the actual function calls** to each implemented function's code.

The entire process is illustrated in Fig. 6.2. An application consisting of four different **modules** is split into multiple files - there are four **header (.h)** files, one for each module. There are also four **implementation (.c)** files with the corresponding implementation. Note that each **module** may use functions from the others, and uses **#include** statements to import the function declarations for those modules during compilation.

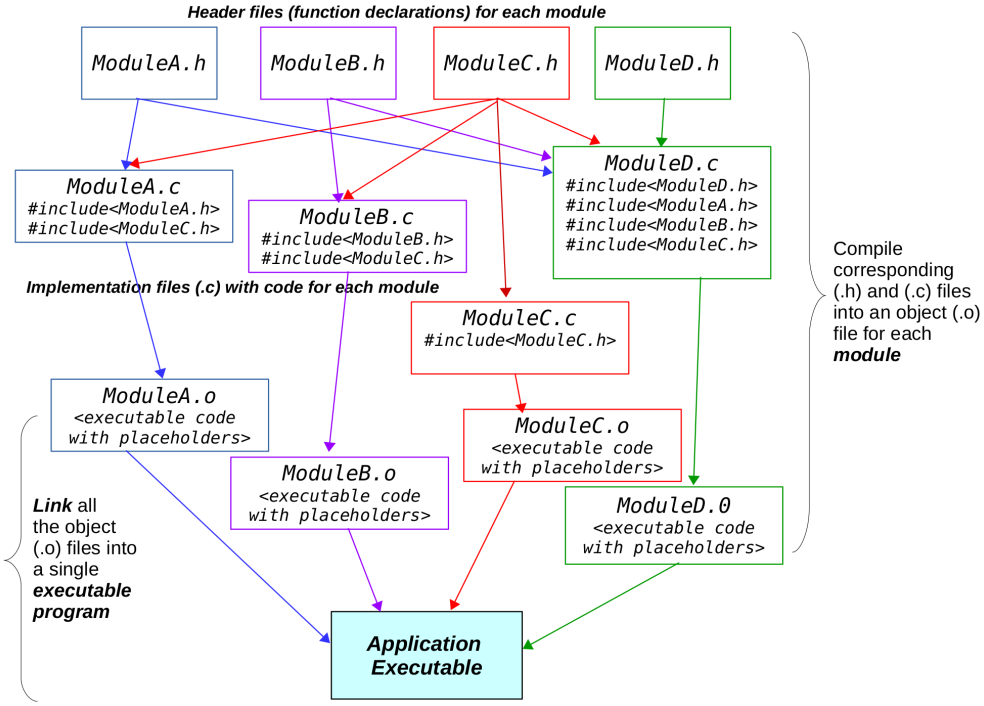

Figure 6.2: The process for combining multiple **modules** into a single **executable program**.

Each **module** is compiled into an **object (.o)** file that contains **executable code** for that particular module's functions. This file contains **placeholders** wherever functions from other modules are used. Once all the modules have been **compiled into object files**, all the **.o** files are **linked together**. This process involves combining the **implementation of any functions used in the program into a single executable file** and replacing the **placeholders** with actual working function calls. The result is a single, working, executable program.

All the **C** programs you have implemented and compiled up to this point had to go through this process. We did not have to think about it because since we have only used standard **C** libraries, the entire process was **done automatically and transparently**. However, once we start developing our own **modules** and developing applications consisting of multiple components, we will need to keep in mind how the different parts work together to build our application.

6.4 Interacting with modules: The Application Programming Interface (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 **provided by our module**, which usually means the set of **functions** other programs can call, and the way in which information will be **passed to and from** between other programs and our module's 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 different **software modules**, **application libraries**, **local or remote computer systems**, and even **internet based applications**.

Note

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

- **Google Maps:** Allows us to make use of the **Google maps** framework for plotting locations and for finding paths between points in maps - among many other things. You can check out the **API** at `https://developers.google.com/maps/documentation/javascript/tutorial`
- **TensorFlow:** Allows us to set up, train, test, evaluate, and operate machine learning algorithms, including **neural networks** for solving a task that require learning from a very large dataset. Nowadays such algorithms are behind some of the most useful applications in A.I. including **Large Language Models (LLMs)**. The **API** can be found at `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. The **API** can be found at `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. The **API** is at `https://docs.unity3d.com/ScriptReference/`

The above is just a tiny sample of the universe of **APIs** out 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 any of the functionality they provide** 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 functions in 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 module 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**. In addition to this (but not necessarily required), there often exists some form of externally maintained **documentation** (e.g. manual pages, a wiki, or a webpage) that describes and summarizes the **API**, and often also provides examples of how to use it.

Let's now see what goes into designing a good **API**, and why it is a challenging but important process worth a significant amount of thought and care.

6.4.1 Why thinking carefully about API design matters

Suppose we are writing a module for graph manipulation that supports finding a path between nodes in the graph. The algorithm is implemented by this function:

```
intList *findPath(int Adj[][], int start, int goal)
{
   /*
      This function returns a linked-list of nodes that from a path
      from:
              start
      to:
              goal

  If no path can be found, the function returns NULL.

      Adj[][] is the adjacency matrix for a graph with size N

   Assumes: Adj(i,j) is 1 if there is an edge from i to j, and 0
      otherwise. i,j are node indexes in [0,N-1]

   intList is a linked list of nodes, each of which has:
      int nodeIndex;
      intList *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 the **input parameters** the function requires in order to do its work.

Thereafter, if we need to use that function in our program all we need to do is:

- Set up an adjacency matrix for my graph with size $N \times N$
- Find the **indexes** of the **start** and **goal** nodes
- Call the function

However, while considering this very straightforward **use case**, we will notice that our **API** is not designed properly - it does not provide a way for us to specify N, the size of the graph. So we need to change our API a bit:

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

With the above change, it seems our API is good to go and we can release this little function for other developers to use. However, as soon as we release the API, another developer comes along who wants to use our function but their **use case** is somewhat different. They need to find a path from a **start** node to **one of a number of possible goal** nodes (for example, to find a path from a current location to any nearby gas station while driving). With the current **API**, the developer is stuck doing something like 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 way to to this, so we now re-define our **API** as follows:

```
intList *findPath(int Adj[][], int N, int start, int goals[k], int k)
{
   /*
      This function returns a linked-list of nodes that from a path
      from:
             start
      to:
             the *first* goal node in goals[] that can be
             reached during search.

  If no path can be found, the function returns NULL.

      Adj[][] is the adjacency matrix for a graph with size N

      goals[] is an array of integers with the index of any
          goal nodes that should be considered by the
          function.

      k is the number of entries in goals[]

   Assumes: Adj(i,j) is 1 if there is an edge from i to j, and 0
      otherwise. i,j are node indexes in [0,N-1]

   intList is a linked list of nodes, each of which has:
      int nodeIndex;
      intList *next;
   */
}
```

Question: Is the above actually better? is the latest version of the **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 this module**.

Example 6.1 After the module is released on **GitHub**, one of the developers who would like to use the module reports that their graph is **too big** and the **adjacency matrix doesn't fit in memory**. They would like to have a way to use an **adjacency list** instead.

That sounds reasonable, but we can not update the function call that uses an **adjacency matrix** to also work with an **adjacency list**. This is a problem we will return 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, we decide we can help developers who have an **adjacency list** by adding one more function to the **API**:

```
   intList *findPath_l(intList* A[], int N, int start, int goals[k], int k)
   {
      This function returns a linked-list of nodes that from a path
      from:
             start
      to:
             the *first* goal node in goals[] that can be
```

```
            reached during search.

If no path can be found, the function returns NULL.

    A[] is the adjacency list for a graph with size N

    goals[] is an array of integers with the index of any
        goal nodes that should be considered by the
        function.

    k is the number of entries in goals[]

 intList is a linked list of nodes, each of which has:
    int nodeIndex;
    intList *next;
}
```

The example above illustrates a common problem when designing **APIs**. Often there are many **small variations** on a problem that users of a **module** or **library** may need support for - depending on their specific task and the rest of their program. However, adding **functions that are small variations of each other** is not a good solution: it creates a lot of **code duplication** since the functions have the same task, but they work on slight variations of the input which means a lot of the code in each function will be identical and yet not easily separable into smaller, independent functions. It also makes the **API** cumbersome since developers now have to worry about figuring out which among the many similar functions they should be using.

Consider what happens if a bug is found in **findPath()** - because there are two versions of it, we will have to fix a problem in two places. The more variants of **findPath()** that we provide for the convenience of developers using our **module**, the more places we will likely need to check and possibly update when problems are found. This makes our **module** more difficult to **maintain** and increases the likelihood we will miss something that needs updating. So it is far from an ideal situation.

This brings us to what happens whenever the **API** is updated. For instance, because **bugs were fixed**. Since this doesn't change any of the function declarations for the **module**, the change doesn't affect how programs using our module are written or how they interact with our **module**. However, in order for the update to take place on programs that use our **module**, every program using it must be **recompiled**. This is illustrated in Fig. 6.3.

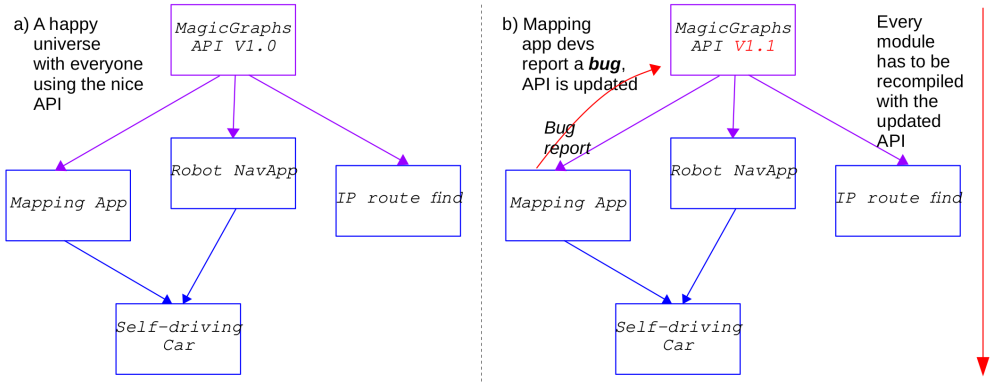

Figure 6.3: Updates to the **module** that do not change the function declarations in the **API** still require that all programs using the **module** be recompiled.

This is not uncommon, and should highlight the importance of having a **solid development and testing process** for any **module** that is going to be made available for use by others - the more users out there for a particular **API**, the larger the impact of any **bugs** or **security vulnerabilities**. This will place a burden on the developers of a successful **API** as they will be expected to stay on top of any **bug reports** or **reported security issues**.

A final issue of note is that once the **API** has been released and it's been picked up by a population of developers, it becomes **very difficult to make changes to it** even if we realize that **it was not designed properly** in the first place. To see why this is the case, consider the following situation:

Example 6.2 A developer who wants to use the path finding **module** requests that the path finding functions support **graphs with weighted edges**. These are very common in all kinds of applications, so it is expected that a potentially large number of people who need a module that does path finding would find this useful.

Unfortunately, this is not something we can easily change without significantly impacting any current users of the **API**. The original functions that perform path finding specify that the edge information is **integer** and only **the presence** or **absence** of an **edge** is indicated.

Weighted edges could have any **real value**, so the **integer** data type will not work in general. To provide the desired functionality, we would have to **change the declaration for findPath()** so that it receives an **adjacency matrix** that stores **float** or **double** values (we would also need to change **findPath_l()** to use an **adjacency list** that stores the **edge weights**).

In terms of updating our **module** and **API** this may not seem too big a change - but it has major implications to any existing users of the **API** all of whom have programs that interact with our original **API**. Because we can not simply **typecast** an **integer array** into a **floating point array**, each of the existing users of our **API** will need to change their program so that their graphs (currently not weighted) are stored in a suitable **floating point array** or **adjacency list**. This is shown in Fig. 6.4.

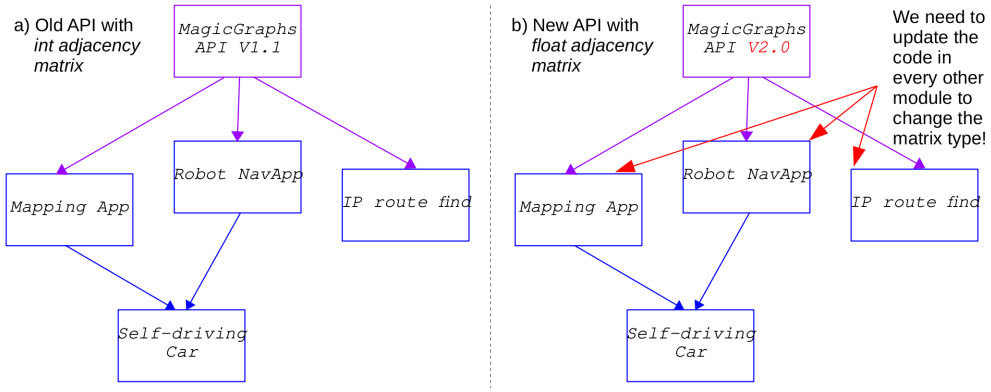

Figure 6.4: Updates to the **module** that **change the function declarations** in the **API** will require all existing users to **modify their own programs** so that they work with the updated **API**. This may involve significant effort, and even if it does not, it is still unwelcome and annoying to developers.

The situation is far from ideal. We have to choose between **making the API more useful to a wider range of developers**, or **avoiding annoyance for existing users**. Note that if we had **thought more carefully** about how to handle **graphs** before writing and releasing our **API** this problem could have been avoided.

We could try to solve the problem by adding **yet another** variant to the **findPath()** function, maybe something called **findPath_d()** which takes a **double precision floating point adjacency matrix**. We may as well go ahead and add **findPath_dl()** which is the equivalent for **adjacency lists**. But now we have **four variants** of the **findPath()** function, with the inherent problem of **code duplication** and the corresponding increase to the work needed to fix **bugs** and keep the **module** up to date.

One last attempt at making everyone happy may involve **simply telling developers how to modify the API themselves** - for instance, by providing them with instructions on how to **patch** their own copy of the **module** so that it supports **weighted graphs**. However, this has two **major drawbacks**:

- Developers using a modified version of the **API** and **module** would be **unable to get updates** unless they are willing to re-do the work of **patching** each update themselves (and this may not even be feasible depending on what was updated).
- If a module using a modified version of the **API** is part of a larger software project that uses other modules that incorporate the **original API**, the larger project **may no longer compile** because of incompatibilities between the versions of the **module** (this is illustrated in Fig. 6.5). There are ways to get around this problem, but they are **band-aid solutions** to a problem that should not exist in the first place, and may not work depending on how different the versions of the **module** become.

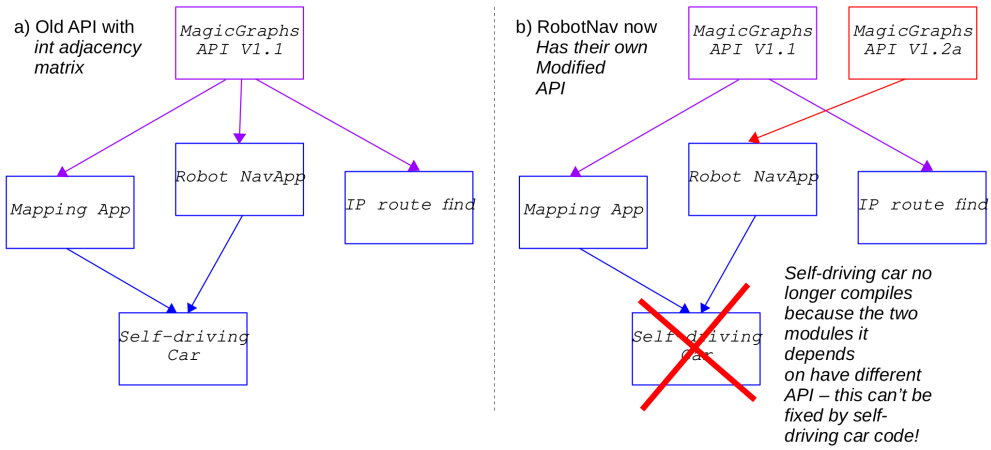

Figure 6.5: Allowing developers to **modify the API themselves** could easily cause larger software projects to break, as incompatibilities between versions of the **API** used by different modules would show up during **compilation** and/or **linking**.

All of the above is to motivate the idea that **we have to be very careful** when we set out to design an **API**, and we have to think very hard about what **use cases** the **module** we are developing may be applied to, and about how to write 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.

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 required to handle, or provide support for.
- **Dependency:** It is a **relation** between software **modules** where a particular piece of software **uses** functionality from, and therefore **requires** that a **library or module** be **available** and have the **correct version** in order work.

In **C**, anything we add to our code via **#include** statements introduces a dependency. For instance, our programs almost always depend on the system libraries **stdio**, and **stdlib** at the very least. If these are not present, we can't compile the program. If they have the wrong version, the executable may not run and may have to be re-compiled with the correct version.

6.4.2 Designing a good API

Advice on how to design a good **API** is best received from those who have the most experience designing some of the most widely used **APIs** currently available. The suggestions and advice below are a summary of the advice provided by **Google** in their article **How to Design a Good API and Why it Matters** which can be found here: 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 us to consider are:

- The **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.
- The **API** should be as **small as possible**, but not smaller. It should satisfy the need it's serving, but should not try to add every single possible thing we can think of. The key idea here is that we can add to it later, but once it's there, it's hard to remove functionality.
- The **API** should be **implementation independent**. This is a particularly important point. It should be possible for us 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** - The **API** should only make available to the user the functionality and data that the user needs. This is harder to do with **C**. 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** must be provided. Every component of the **API** 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. It should be clear what the error is, and where it 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.

There are also a couple of suggestions about good programming practice that are not specific to the design of good **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 make sure to revisit the advice there as you become familiar with object oriented programming. 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 we can never achieve perfection.

6.5 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 are providing a **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). We have determined the appropriate **API**, and written both our (very well documented) header file, and the corresponding implementation file. Any user needing a linked list can include our module in their code and have all the functionality of linked lists at their disposal. However, **there is a catch** - any programs that use a linked list provided by our **module** must have access to the **head of the list**, so the pointer to the head of the list has to be declared and is owned by **code outside of the module**. The situation is illustrated in Fig.6.6.

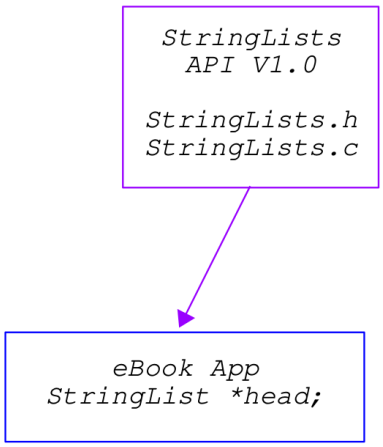

Figure 6.6: A **module** that provides functions to handle **linked lists of strings**. Unfortunately, **the head pointer** has to be declared and is owned by **code that is outside the module**.

This is not great - the user of our **module** has access to all the **CDT definitions** from the **module** (which it needs in order to be able to declare and use the head pointer), and could easily create its own **nodes, pointers, and even lists** without using the **API** functionality. Worse than that, the code outside the **module** has access to all the information stored in the **linked list**, and could easily access, modify, or even delete any of the data stored there without going through **API**. This is a problem because it allows users whether by mistake, or intentionally, to do things with information that the **module** should be handling that were not intended and are not provided by the **API**.

The situation above breaks the concept 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 **security and privacy** concerns because there is no way to **hide** or **protect** information that should not be available outside of the **module**.

The root cause of this problem is that **C** provides no mechanism for **protecting** sensitive data from being misused or accessed in a way that was not intended by us when we developed the **module**. We can't solve this in **C** without severely impacting the usefulness of our **module**. So at this point we have to 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**. We have to be able to **control access** to the data managed by the **module** in a way that prevents users from unintentionally, or intentionally, accessing anything we did not intend for them to have access to. This is called **information hiding**, and is one of the fundamental principles of good software design.

6.6 Object Oriented Programming (OOP)

Object Oriented Programming is built around two principles: **Encapsulation**, and **information hiding**.

Definition 6.1 (Encapsulation)

Encapsulation means **wrapping together** *all the components required to implement the functionality of particular* **software module**. *This includes* **all the data** *as well as* **the functions** *that manipulate it.* **Object oriented** *programming languages must provide support for* **encapsulation** *as a central feature of*

the language (as opposed to something that can be achieved by being creative with language features not designed to provide encapsulation).

♣

Definition 6.2 (Information hiding)

This refers to the ability of the designer of a **software module** *to decide* **which functionality and data** *should be* **visible** *to a user of the* **module**, *and to* **hide** *everything else. This includes* **implementation details**, **data that is essential for the correct working of the module**, *and* **functionality** *that is part of how the* **module** *gets its work done but should not be directly accessed by a user.* **Object oriented** *programming languages provide support for* **information hiding** *by allowing a designer to* **control access** *to the* **data** *and* **functions** *that are part of a* **module**.

♣

To support **encapsulation** and **information hiding**, **Object Oriented Programming** introduces the concept of an **object** as the fundamental unit of **functionality**, which includes **data storage**, as well as **information processing**. We will now spend a bit of time understanding how **objects** 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.

6.6.1 Classes

In **Object Oriented Programming**, we expand on the idea of **compound data types** that we discussed in Chapter 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 6.7.

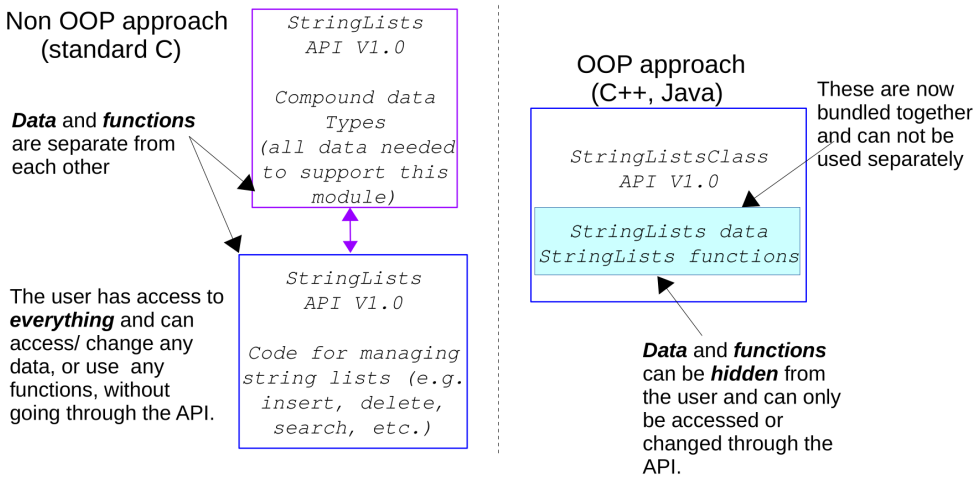

Figure 6.7: A comparison between **non-object oriented** model (a **CDT** plus **separate implementation**) versus an **object oriented** model. In the **OOP** model, everything is **bundled** into a single **class**.

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 **CDTs**, 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 are very difficult (or even impossible) to implement without the support of **object oriented** languages, and that allow us to build software that is conceptually easier to understand, maintain, expand, test, and debug.

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

6.6.2 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**, in addition to their **data type**, both the **member variables** and the **class methods** have an associated **access control modifier**. 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 accessed by (is visible to) any code whether that code is part of the **class** or whether it belongs to an **external program** using the **class** to do some work. It is appropriate for all the components of the **class** that the user will need to call in order to **use the class** for its intended purpose.
- **private** - This modifier states that a **member variable or class method** can be accessed by (is visible to) exclusively the code that is part of the **class methods**. No external program can **see**, **access**, or **use** any of the **private data** or **methods**.

There are further modifiers that may or may not be available depending on the language we are using, but the two above constitute the minimum subset that will allow us to implement **information hiding**. The diagram in Fig. 6.8 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** methods are only accessible from within the **class**.

In the example from Fig. 6.8, the **head pointer** is not visible to the user, **it can not be changed from outside the class**. However, functions such as **insert()** which are **class methods** 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 example from Fig. 6.8: the **constructor** and the **destructor**. These two functions have an important role for the **class**:

- **The constructor** - Is a **method** that is **automatically called** by the when we **create** a new **instance of the class** in a program. The **constructor** has the job of initializing the **member variables** and **any data** that the **class** will need to suitable values. In the example on Fig. 6.8, 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.

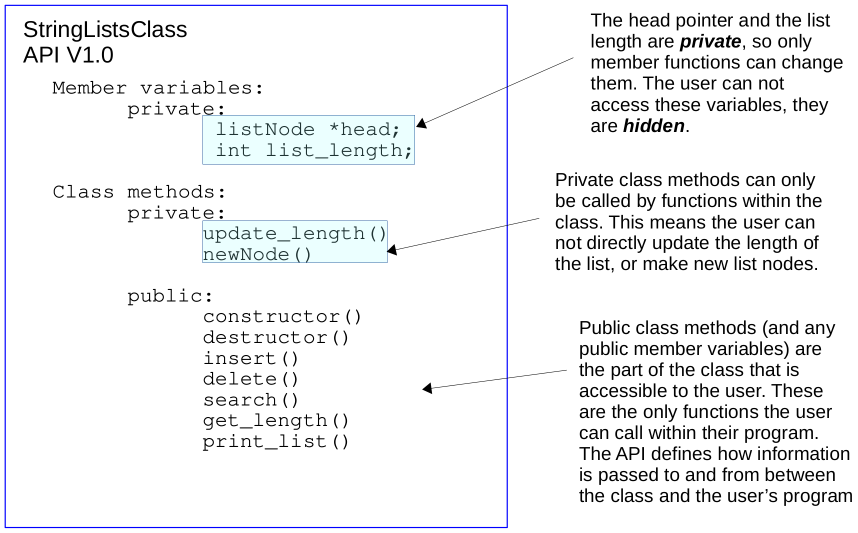

Figure 6.8: Example of **class** structure showing components with different **visibility**. Any part of the class declared as **private** will only be **accessible** from within the **class** itself. Any **public variables and methods** are **accessible** from any code where the **class** is being used.

- **The destructor** - Is a **method** that is **automatically 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 the example from Fig. 6.8, 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** work so the user of the **class** doesn't have to do it themselves.

Note

Just what is an object? In **non-object oriented** programming, we create **variables** out of any **CDT**. We simply call them **variables** and think of them as **units of data**. The equivalent in **OOP** is a **class instance** or for short **an object**. This is where the **object** part of **object oriented programming** comes from.

The **class** is the template for building **objects**, just like the **declaration of a CDT** is a template for declaring **variables** of that particular **compound data type**.
Each **object** should be thought of as a **big box in memory** that has all the **member variables** and **class methods** specified by the **class declaration**. **Objects** are the essential **functional unit** in **OOP**. The difference between a **class** and an **object** is illustrated in Fig. 6.9.

6.7 Implementing a class in C++

In order to explore some of the features of **OOP** that help us design and implement good software, we need to see how these features work in the context of an actual **Object Oriented** programming language. For this chapter

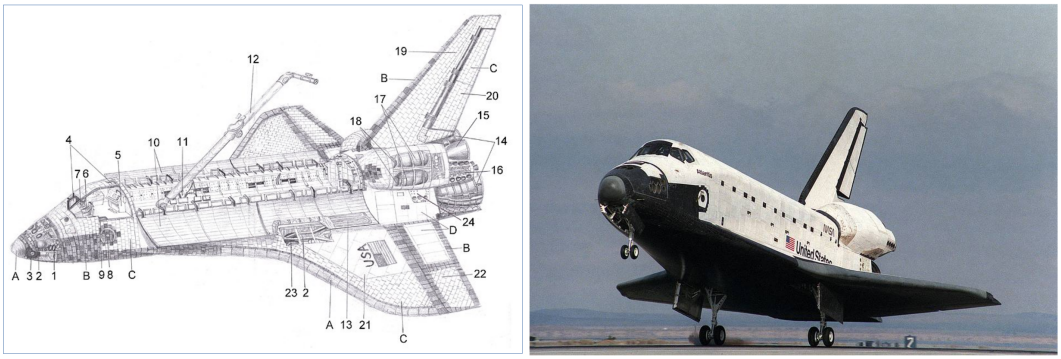

Figure 6.9: The **class** is a blueprint for building **objects**, the **class** specifies all the parts, components, and functionality each **object** must have. From the class, we create **objects** which are specific **instances** of a **class**. *Images: Left-side, Wikimedia Commons, by Eryn Blaireova, CC-SA2.5. Right-side, U.S. National Archives, Public Domain.*

we will use **C++**. This will allow us to apply all the work we have done thus far in learning **C** as we work on developing an understanding of how **object oriented programming** allows us to build software in ways that were not possible with regular **C**.

C++ was developed in the 80's as a significant improvement over standard **C**. It was provided with the syntax and features 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 software development. Areas such as operating systems, security, networking, embedded systems, media-related software (e.g. encoding/decoding video and audio), and compilers; among others make extensive use of software written in **C++**. Because it is developed from **C**, the basic building blocks of **C++** programs are already well known to us. Function declarations, variables and standard data types, loops and program control structures are all the same as in **C**. All the regular **C** programs we have written thus far are also valid **C++** programs. But the range and flexibility of features provided by modern **C++** compilers far exceeds anything regular **C** can do.

With that in mind, let's see how a **class** is declared in **C++**, and consider the similarities a **class** declaration has with regard to **compound data types** in **C**. The following listing could be stored in a **header** file called **StringList.h**

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

// Notice the #include statements above. We are using standard
// C libraries (which is allowed since C++ compiles any standard
// C code} for familiarity.
// However, if you want to write pure C++, you should use the
// object oriented libraries that provide much more advanced
// (and object-based) functionality! We will not do that here
// because this is not meant to be a book on C++ programming.
// Here is a good-ol' CDT to store nodes in a
// linked list - for comparison with the class
// just below. There is nothing fancy in the CDT,
// just a bento box with data in it.

typedef struct ListNodeStruct
{
```

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

// Here is something new. A StringList class,
// which bundles together all the variables
// and data needed for the list, as well as
// the functions that implement the functionality
// required of the list of strings.
class StringList
{
  // Member variables
  private:
              ListNode *head;
              int list_length;

  // Class methods
  public:
              StringList()
              {
                head=NULL;
                list_length=0;
              }
              ~StringList()
              {
                  clear_list();
              }

              void       insert_string(char string[1024]);
              void       delete_string(char string[1024]);
              void       print_strings(void);
              ListNode   search(char string[1024]);
              void       clear_list();
              int        get_length();

    private:
              void  remove_head(void);

};
```

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 are building.
- The **class declaration** is just like any other **CDT** declaration we have done before in **C**, except we use the keyword **class** so the **C++** compiler knows we are **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()** is the **class constructor**. It 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.
- 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()**.
- 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.

In order for the class to be useful, we need to provide an implementation for the **class methods**. The implementation typically goes into a separate program file with the extension **.cpp**. Here is what the implementation looks like for the short class we declared above:

```
#include "StringList.h"

// Notice the 'StringList::' prefix for function declarations.
// This is a class name qualifier and tells the compiler we are
// providing an implementation for a function of the 'StringList'
// class. Without it, the compiler would assume we are
// implementing a regular (non-class member) function that
// happens to be called 'insert_string()'.
// All functions that are part of the class must have the
// corresponding qualifier.

void StringList::insert_string(char string[1024])
{
    // Insert a new string into the linked list
    // at the head of the list (for simplicity)

    ListNode *p=new ListNode;    // This is how we dynamically
                                 // allocate data on-demand in C++
                                 // (no need for calloc)

    if (p==NULL)
    {
        printf("There is no memory for more strings!\n");
        return;
    }

    strcpy(&p->string[0],string);

    if (this->head==NULL)        // 'this' is a pointer to
    {                            // the specific object for
        this->head=p;            // which the function was called
        this->list_length++;
        return;
    }
    else
    {
        p->next=this->head;
        this->head=p;
        this->list_length++;
        return;
    }
}

void   StringList::delete_string(char string[1024])
{
       // Deletes a string from the linked list if it
       // is found there
```

```
        ListNode *p, *q;

        p=this->head;
        // If requested string is at the head of the list
        if (strcmp(p->string,string)==0)
        {
            this->head=this->head->next;
            this->list_length--;
            delete p;
            return;
        }

        // Otherwise
        while(p->next!=NULL)
        {
            q=p->next;
            if (strcmp(q->string,string)==0)
            {
                p->next=q->next;
                delete q;
                this->list_length--;
            }
            p=q;
        }
}

void StringList::print_strings(void)
{
    // Print out all the strings currently in the list

    ListNode *p;

    printf("*** The strings currently in our list are:\n");
    p=this->head;
    while(p!=NULL)
    {
        printf("%s\n",p->string);
        p=p->next;
    }
}

ListNode StringList::search(char string[1024])
{
    // Find a node in the list that contains the requested
    // string. Return a *COPY* of the node that has no
    // pointers to list data. If no matching string is found,
    // it returns an <empty> ListNode.

    ListNode *p, copy;

    strcpy(&copy.string[0],"");
    copy.next=NULL;

    p=head;
    while (p!=NULL)
    {
        if (strcmp(string,p->string)==0)
        {
            copy=*(p);
            copy.next=NULL;
```

```
            return copy;
        }
        p=p->next;
    }

    return copy;
}

void    StringList::clear_list(void)
{
    // Free all memory allocated to the linked list, reset
    // length to zero, and set head pointer to NULL
    while (this->head!=NULL)
      remove_head();
}

int     StringList::get_length(void)
{
    // This is what in object-oriented programming is called
    // a 'getter'. A function that returns the *value* of
    // a private variable so the user can find out what it is
    // without actually getting direct access to read or change
    // the variable.

    return this->list_length;
}

void    StringList::remove_head(void)
{
    // Removes the node at the head of the linked list
    // and releases the memory allocated to the list
    // node. It reduces the length of the list by 1
    //
    // This function is used by the function that
    // clears the linked list. We do not give the users
    // of the class access to it because they could
    // use it to remove list data without
    // regard for the contents. Hence, we make this
    // function private.

    ListNode *p;

    if (this->head==NULL) return;

    p=this->head;
    this->head=this->head->next;
    delete p;
    this->list_length--;
}
```

A few **syntax** notes are worth a quick look:

- The declarations for the **class methods** have to be preceded by the name of the **class** and a '**::**'. This is used by the compiler to figure out which **class** each function is a part of (there may be many different **classes** with functions that have the same name, or there can be functions that are not part of any **class** that have the same name as a **class member**.
- We no longer have to use **malloc()** or **calloc()** to **dynamically allocate memory**. **C++** provides a keyword

new that allocates and initializes new memory on request, as well as a corresponding **delete** keyword that releases space once we are done using it. These two keywords exist because with objects some extra work has to be done. Whenever we create a new **dynamic object**, the **constructor** has to be called. The **new** operator does this. Similarly when we are done using an object and want to release its memory, the **destructor** has to be called. The **delete** operator takes care of that.

- To access any of the **variables or class methods** from **within the class member functions**, we use the **'this'** pointer. It works just like any other pointer in **C** and **C++** and we can get to any **component of the class** by using the **-> (arrow)** operator. What's up with **'this'**? If we think about it, to access a **field** in a **CDT** we use **variable_name.field_name** (or **pointer_name->field_name**). However, **we do not know the name the object will have in advance**. The user will call their **objects** whatever they want, and there will most likely be many **objects** with different names from the same **class** - so we can't use the **object's name** while writing the implementation of the **class member functions**. The **'this'** pointer is a convenient way for the compiler to know that whatever name the user gives an **object**, a **member function** needs access to a specific **field** within that particular **object**.

At this point, it's worth spending a moment or two thinking about how the **StringList class** we just created compares to the **non-object oriented** linked lists we developed in Chapter 3. The fact that both the **data** and **functionality** are **bundled** together as part of the **StringList class** makes the design of the software **cleaner**, **more intuitive**, and **easier to understand**. Much like **CDTs** allowed us to represent complex data items as **individual units of information**, **classes** allow us to represent software components as **individual units of functionality**.

Not only that, with our original linked lists written in **C** there was no protection against a user of the code accidentally (or maliciously) accessing sensitive information (such as, for instance, the pointers that keep the list organized and properly linked). The **StringList class** uses **information hiding** to ensure that users have no access to such sensitive **variables** and/or **class methods**. To understand how this works, let's look at a very simple program that creates **one object** of the **StringList class**, and then tries to access **private variables** or **private class methods**.

```
#include "StringList.h"

int main(void)
{
    // This is a very short example of using the StringList library to
    // create a linked-list of strings.

    StringList my_list;      // Here is an actual object of the
                             // StringList class.

    // Let's see what happens if we try to access private variables or
    // methods:

    printf("The length of the list is: %d\n",my_list.list_length);

    // Try and call a private class method
    my_list.remove_head();

    return 0;
}
```

If we try to compile the program above, we will see the following:

```
> g++ -c stringExample.cpp
stringExample.cpp: In function 'int main()':
stringExample.cpp:14:54: error: 'int StringList::list_length' is private within this context
     printf("The length of the list is: %d\n",my_list.list_length);
                                                      ^~~~~~~~~~~
In file included from stringExample.cpp:1:0:
StringList.h:33:21: note: declared private here
                int list_length;
                    ^~~~~~~~~~~
stringExample.cpp:17:25: error: 'void StringList::remove_head()' is private within this context
     my_list.remove_head();
                         ^
In file included from stringExample.cpp:1:0:
StringList.h:55:23: note: declared private here
                void remove_head(void);
                     ^~~~~~~~~~~
```

The compiler will throw an **error** any time that user programs attempt to access **private data or methods** contained in an object. There is **no way** to build a working executable program unless these errors are resolved by removing any attempts to access **private** components of the **class**. In this way, the compiler enforces our decision to **hide certain parts of our class** from users of the **module**.

Let's now see a short example of a program using our **StringList class** as intended, to store and manipulate strings:

```
#include "StringList.h"

int main(void)
{
    // This is a very short example of using the StringList library to
    // create a linked-list of strings.

    StringList my_list;      // This is is an actual object (and instance)
                             // of the StringList class.

    printf("Adding a couple of strings to the list...\n");
    my_list.insert_string("First String");
    my_list.insert_string("Second String");
    printf("The length of the list is %d\n",my_list.get_length());
    my_list.print_strings();

    printf("Add one string and delete another...\n");
    my_list.insert_string("Third String");
    my_list.delete_string("First String");
    printf("The length of the list is %d\n",my_list.get_length());
    my_list.print_strings();

    printf("Clear the list of strings...\n");
    my_list.clear_list();
    printf("The length of the list is %d\n",my_list.get_length());
    my_list.print_strings();

    return 0;
}
```

The listing above should not be too surprising. But it illustrates just how **clean**, **intuitive**, and **easy to follow** a

program becomes when it uses **objects** to do its work. The equivalent **C** code would have much longer and more cumbersome function calls (which require additional parameters) and would be **longer** and **less tightly integrated** than the **object oriented** program shown above.

As a very simple example, consider the task of figuring out the **length of the list**. In the **C** implementation there was no easy way to keep track of this value, and we had to do a **list traversal** to count the number of entries every time we needed to find the length. In the **StringList class** this is easily resolved by having a **member variable** that keeps track of the length, and that is directly manipulated by functions that insert or remove entries from the list. This removes the need for a **list traversal**.

Compiling and running the code above produces the following output:

```
> ./a.out
Adding a couple of strings to the list...
The length of the list is 2
*** The strings currently in our list are:
Second String
First String
Add one string and delete another...
The length of the list is 2
*** The strings currently in our list are:
Third String
Second String
Clear the list of strings...
The length of the list is 0
*** The strings currently in our list are:
```

6.7.1 Method Overloading

Recall that as we were designing a simple **API** for path finding, we ran into a situation where one some of the users of our **module** wanted to use an **adjacency list** instead of an **adjacency matrix**. In **C** there is no clean solution that allows users to use either of these (as needed) to access the functionality provided by our **module**, and we were stuck creating multiple functions with similar names and **significant amounts of code duplication** in order to provide the needed functionality. Code duplication is not a good thing and should be avoided whenever possible. We will soon see ways in which **object orientation** allows us to elegantly expand and refine an **object's** functionality without unnecessary code duplication. But first, let's look at a feature of **object oriented** languages that allows us to avoid having multiple functions with similar names all of which are intended to provide the same functionality.

In **C++** and other **object oriented** languages, we are allowed to declare multiple functions **with the same name but different arguments and/or return value types**. This is called **method overloading**.

In our **API** example, we had a situation where some users needed a function that worked with **an integer adjacency matrix**, some users wanted a function that worked with a **floating point adjacency matrix** so they could use it on **graphs with weighted edges**, and some users wanted a function that worked with **an adjacency list** (we decided we should also support **weighted** and **non-weighted edges**). This created four functions with different names in **C**. With **C++** the situation is different, we are allowed to declare the functions as follows:

```
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);
```

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

This may not seem like a big change - after all we still have **four functions** with **slightly different functionality** which all do the same thing. However, they all have the **exact same name** which makes it **conceptually easier** for a developer to work with them. **Method overloading** makes the code for programs using a **module** easier to read, and allows an **API** to provide the functionality required in a **clean** and **concise** way. With an **object oriented** language, there is no possible confusion between **functions that have similar names but provide different functionality**, and **variations of the same function which have different names because the language doesn't support overloading**.

How does **method overloading** work in practice? The compiler has access to all the different variations of the function, when a user program calls the function, the compiler **checks the list of arguments passed to the function** and **finds a matching definition** among the **overloaded methods** we provide in our **module**. If a matching method is found, the compiler will use the correct function all. Otherwise it reports that there is no matching method for the specific list of parameters the user provided.

Just a word of caution: While we can provide as many **different overloads** for any given **method** as we want, we should not simply **create code blot** by providing every possible combination of parameters we can think of. Instead, every **version** of an **overloaded method** that we decide to add to the **API** should be supported by a valid **use case**.

Note

Method overloading is an example of what in **object oriented programming** is called **polymorphism**. **Polymorphism** is a term for a thing that has multiple different shapes. In the case of **method overloading**, it refers to a function that has the same purpose (same functionality, same name) but different **lists of parameters and/or return type**. We will soon see that **polymorphism** also occurs at the level of **objects** and that it provides us with incredible power in terms of **building functionality** into our **classes**.

6.8 Inheritance and class hierarchies

A very common situation in software design involves writing programs that can work with a variety of data items that are **variations, or refinements of each other**. For example, software to run a library catalog will need to handle records for **books, magazines, journals, periodicals, graphic novels** and other media types. These are all related, they have some **common properties** (e.g. a name, a date of publication, a library code), and some **different attributes** that are specific to each particular type of item. As another example, consider a **media player** application - it is required to handle a large variety of **media types** and **media formats**. The corresponding **media files** have **common properties** as well as **attributes that are specific to each media type**.

Handling such situations without **object oriented programming** results is code that is **cumbersome**, **hard to maintain, test, debug, and expand**, and that contains **significant amounts of code duplication**. To see why this is the case, consider the following concrete problem:

Example 6.3 We are implementing a **software synthesizer**. This is a program that takes a music file that contains **information about notes in a song**, the **timing and duration of each note**, and also **the instrument that plays**

each note. The **synthesizer** then **generates sound** to play the song. One way to think of it is this: Give the **software synthesizer** a **musical score**, and it will **play the music** shown in that score. It is **not the same as a recording** such as you would get from a streaming service or store on your smartphone - they key is that **the user can change the score** and the music will change accordingly.

A key component of the **software synthesizer** is a **list of notes** that are playing at a particular time. Each of these is then processed to generate the corresponding sounds. In **C**, we would normally represent a note as a **CDT** using something like this:

```
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
                              // should be played with
    double *note_data;        // Pointer to the note's data array
} Note;
```

But if you think about it, this is very **limiting**. Different instruments may require additional information in order to play a note properly. For example, if a note is being played by a guitar, we may need to know which of the guitar's strings is used to play it (which will change the sound). If in addition the note is being played by an electric guitar, we may need to know the state of the **distortion pedal** or the **wah-wah** lever. The point is, **if we are using CDTs**, and we need the software to work with **notes** as a fundamental unit of information, we would need to pack **every single piece of information that any of the instruments may need** into the same **CDT** - there is no way to **modify or adapt** the **CDT** for each instrument. The resulting **CDT** will look like so:

```
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
                              // should be played with
    int    stringNo;          // Which string is used (for guitar only)
    double pedal_val;         // State of the pedal (for electric guitar only)
    double wahwah;            // State of the wah-wah level (electric guitar only)
    double soft_p;            // State of the soft pedal (piano only)
    double damper_p;          // State of the damper pedal (piano only)
    .
    .                         // Many, many more variables needed by each of the
    .                         // different instruments
    .
    double *note_data;        // Pointer to the note's data array
} Note;
```

This is not a very elegant solution, and also wastes a significant amount of storage space, as most notes will use only a handful of the variables contained in the **CDT** to generate their sound.

Having a big and bulky **CDT** is not ideal, but it is not our only problem. Somewhere in the **software synthesizer** code there will be a function whose job is to generate the actual sound data for a particular note. You can imagine

a **loop** that goes over each of the notes that need to be played at some point in time, and for each node, it calls the function that actually produces the corresponding sound. The sound producing function will look as shown below:

```
double get_sound_sample(note *my_note, double time_index)
{
    // Computes and returns the sound value for the specified note
    // and time index.

    if (note->instrument_ID==0)
    {
        // Do the processing required to get a sound value
        // for instrument 0, using the required variables
        // from the CDT
    }
    else if (note->instrument_ID==1)
    {
        // Do the processing required to get a sound value
        // for instrument 1, using the required variables
        // from the CDT
    }
    else if (note->instrument_ID==2)
    {
        // Do the processing required to get a sound value
        // for instrument 2, using the required variables
        // from the CDT
    }
    else if ...    // for as many instruments as the synthesizer
                   // supports (this can be hundreds!)

}
```

Within each **if...else** block, there will be code that produces sound for a specific note and specific instrument. A significant portion of this code **will be identical**, so there is a lot of code duplication going on here. Beyond being **long, cumbersome, and hard to read** for a developer, we have a serious problem in terms of **maintaining, expanding, testing, and debugging** this function. With so much duplicated code, if we find a **bug** chances are we will have to apply fixes (and then test them) at multiple places in this function. This increases the likelihood we will miss something. Expanding the functionality of the function (e.g. to add another instrument) only makes this situation worse.

In a situation like this, it is not difficult for the code to become **unmanageable**, **unmaintainable**, and eventually **stale** as the effort required to keep it up to date, fix bugs, and add functionality becomes too great.

We would like to solve a problem similar to the one we just described above, but at the same time we need to:

- **Maximize code reuse** - Any code that is shared among the different variations of our items should be implemented only once.
- **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 items without affecting the rest.
- We want our resulting software to be organized in a way that makes the relationship between our data items clear, and easy to understand conceptually.

6.8.1 Hierarchies of objects

With **object orientation**, we can build programs that take advantage of the idea that **related objects** can be organized into **hierarchies** in such a way that **shared characteristics and functionality are not duplicated** but we can, at the same time, **provide any level of refinement** that we need for each of the **different objects** in the **hierarchy**.

For our **software synthesizer** and the various types of **notes** it has to handle, a possible hierarchy would look like the diagram shown in Fig. 6.10.

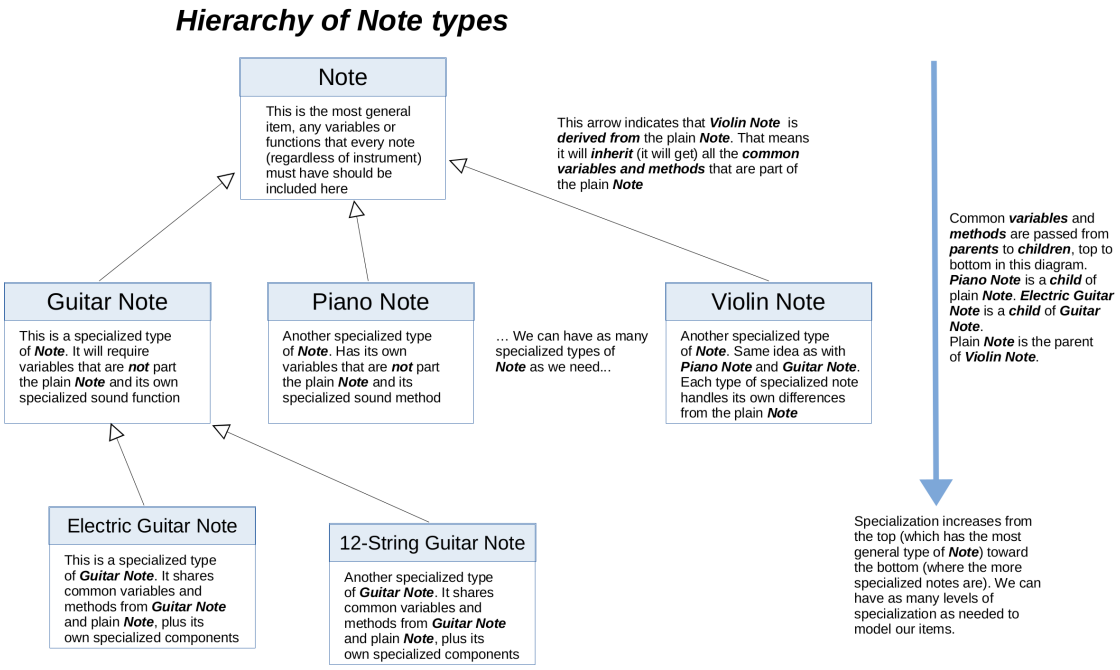

Figure 6.10: A diagram illustrating the hierarchical relationships between different types of **notes**. The idea being that we can build functionality by starting with more general (abstract) items, and then creating multiple **specialized** versions of them that share **common attributes**.

The diagram illustrates the notion that there are **attributes** (**variables and functionality**) that all **notes** must have. It makes sense to bundle them together into a plain **Note**. We can then **derive** different versions of the plain **Note**, one for each of the instruments our synthesizer supports. The **specialized** note types will **inherit** all the **common attributes** of their parent (the plain **Note**), and add their own **specialized attributes** that are required to generate their particular type of sound. We can create as many levels of specialization as we need, so for example we can **derive** various specialized types of **Guitar Note**, or **Piano Note**, or **Violin Note** in order to further enrich our synthesizer's capabilities.

Because our different **notes** are organized in this hierarchical fashion, there is **little or no duplication** of either **data** or **functionality**. Each **specialized** type will define **only** those **attributes** that are **not shared** with other types of **note**, and that make each specialized type unique. Any **common attributes** are created once and then shared according to the relationships described by the hierarchy.

6.8.2 Implementing hierarchies and inheritance

Object oriented languages allow us to implement hierarchies like the one shown in Fig.6.10 by allowing us take any **class**, and **derive specialized versions** of it. The **derived classes** are often called **child classes**, while the original **class** is usually called either **base class** or **parent class**. Let's see how that would work in the example of the **software synthesizer** and see how the use of **class hierarchies** results in a program that is much **cleaner**, and easier to **read, maintain, test, and debug**.

Example 6.4

One of the first step in designing the **software synthesizer** would have been to **figure out the hierarchy of object types** that we need to handle. This is not always straightforward and should receive careful attention because once the hierarchy is implemented, it can be difficult to change it in a significant way. So as part of our design process (as described in Chapter 2), with **OOP** we have to spend a good amount of time figuring out how to organize our data and functionality into **classes** and **class hierarchies**, and we need to figure out **which attributes will belong in which class** within the hierarchy. This work is well worth the effort, as a well designed **class hierarchy** will make our work all that much easier when implementing, testing, and debugging our software.

For this example, we will work with the hierarchy shown in Fig. 6.10 and implement the plain **Note** as well as **two child classes**, the **Guitar Note** and the **Piano Note**. This is simply to illustrate how the process works and the principles involved in building and using a simple class hierarchy. The example can easily be extended to incorporate more **child classes** at different levels of specialization.

Let's have a look at the implementation of the **Note class** which is the **parent class** for both **Piano Note** and **Guitar Note**.

```
class Note
{
 // Member variables
 protected:
              double time_position;   // When the note starts
              double frequency;       // The note's frequency
              double duration;        // Duration in milliseconds
              int volume;             // Volume in 0-10
              double *note_data;      // Data array for the note

 // Class methods
 public:
              Note()
              {
                // Default constructor, used when we do something
                // like this: my_note = new Note;
                this->time_position=0;
                this->frequency=0;
                this->duration=0;
                this->volume=0;
                this->note_data=NULL;
              }

              Note(double t, double f, double d, int v)
              {
                // A constructor that initializes the various values
                // for this note, used when we do something like
```

```
    // this: my_note = new Note(1.25,440,1.5,7);

    // It is an example of member overloading. The
    // compiler chooses the right constructor based
    // on the parameters we specify while creating the
    // Note!

    this->time_position=t;
    this->frequency=f;
    this->duration=d;
    this->volume=v;
    this->note_data=NULL;
  }

  virtual ~Note()
  {
    printf("* Note class destructor called!\n");
    // Release memory for this note!
    if (note_data!=NULL) delete note_data;
  }

  virtual double get_sound_sample(double time_idx)
  {
      // This function would generate the
      // correct sound value for a plain Note
      // (maybe sounds like a beep?) at the
      // specified time index, given the
      // note's frequency, duration, and volume.

      // Here, for simplicity, it will simply
      // print a message

      printf("A plain Note with frequency %f is making sound at time %f\n",this->
          frequency,time_idx);
      return 0;
  }

  virtual void print_note_info()
  {
      // Prints out the values of the note's
      // variables
      printf("This generic Note has frequency %f\n",this->frequency);
      printf("This generic Note has duration %f\n",this->duration);
      printf("This generic Note has volume %d\n",this->volume);
      printf("This generic Note plays at time index %f\n",this->time_position);
  }
};
```

The **Note class** contains all the **attributes** that are common across **all the possible types of notes** that our synthesizer may need to handle. This includes a few key **member variables**, and **two methods** that are common to all notes. A couple of notes on syntax:

- The **protected** access modifier is something we didn't see before. It allows us to **share common attributes** amongst **objects** that are part of a **class hierarchy** while preserving **information hiding**. Specifically, **protected** components of a class are **visible** within code in the **derived classes** but are **hidden from code that is not part of the parent or derived classes**. As far as code **outside the classes** is concerned, **protected**

components behave as if they were **private**. We do not use the **private** access modifier here because any **private** attributes would **not be visible** within **derived classes** (notice that they still can be used by a derived class object through any inherited methods implemented in the base class). Having both **private** and **protected** access modifiers allows us to have very fine control regarding how **class attributes** are **shared or not** across a hierarchy, and at the same time enforce **information hiding**.

- There are **two constructors**, the **default constructor** which takes no parameters and initializes the **class**' attributes to default values, and a **constructor** that accepts a list of initialization values for the **data members** of the **class**. This is an example of **method overloading** and is very common - often we want to provide different **constructors** corresponding to different ways in which users may need to create **objects** of a given **class**.
- The **get_sound_sample() method** is declared as **virtual** - this tells the compiler that we expect this function to be **specialized** by the **derived classes**. This doesn't mean that **child** classes are forced to re-implement the function, it simply means that they **could** and **often will** do just that. If the **derived class** provides their own version of **get_sound_sample()**, then their **specialized** version will be used. Otherwise the implementation from the **Note** class will be used.

In the listing above, we created a very simple *dummy* function to generate sound samples. It doesn't actually make any sound, it prints a message instead - we will use this in a moment to see how the class hierarchy works, but in a real implementation there would be a whole lot of code in this function, related to computing and returning the appropriate sound value for the **Note** at the specified time index.

Now that we have the **parent class**, let's see how we would create the two **child classes**: **Piano Note** and **Guitar Note**:

```
class PianoNote : public Note
{
    // This is a derived or child class. The parent is 'Note'
    // and the 'public' access modifier states that:
    // All 'public' attributes of 'Note' will be 'public' in PianoNote
    // all 'protected' attributes of 'Note' will be 'protected' in PianoNote

    protected:
            double soft_p;    // State of the soft pedal
            double damper_p;  // State of the damper pedal

    public:
            PianoNote():Note()
            {
                // Default constructor - it calls the default constructor of the parent class!
                this->soft_p=0;
                this->damper_p=0;
            }

            PianoNote(double t, double f, double d, int v, double sp, double dp) : Note(t, f, d,
                 v)
            {
                // Constructor with initialization data, calls the corresponding constructor
                // in Note for initializing common attributes.
                this->soft_p=sp;
                this->damper_p=sp;
```

```
        }

        ~PianoNote() override
        {
            // Should do anything needed to clean up after
            // PianoNote information *but* not the common
            // information from 'Note', since the destructor
            // for 'Note' will be called immediately after this
            // function is done.
            // So, here, we don't need to do anything at all...
             printf("** PianoNote class destructor called!\n");
        }

        double get_sound_sample(double time_idx) override
        {
            // This is the implementation of the get_sound_sample
            // which should be different from a PianoNote. Here
            // we will simply place a print statement to indicate
            // this function is being called

            printf("A PianoNote with frequency %f is playing at time index %f\n",this->
                frequency,time_idx);
            return 0;
        }

        void print_note_info() override
        {
            // Prints information for this PianoNote
            printf("This PianoNote has frequency %f\n",this->frequency);
            printf("This PianoNote has duration %f\n",this->duration);
            printf("This PianoNote has volume %d\n",this->volume);
            printf("This PianoNote's soft pedal is at %f\n",this->soft_p);
            printf("This PianoNote's damper pedal is at %f\n",this->damper_p);
            printf("This PianoNote plays at time index %f\n",this->time_position);
        }

};
```

Consider how easily we created a new type of **note**. We used the **common attributes** from **Note**, and simply added the **data members** that are specific to the **PianoNote**. Just two variables, because **PianoNote** will inherit **frequency, duration, time_position, and volume** from **Note**. We are actually using those **common attributes** within the implementation for **PianoNote**, for instance, in the function that prints the values of the **PianoNote** variables. But we don't have to declare them within the **PianoNote class**, they are passed down from the **parent** class.

Notice as well that the **constructors** for **PianoNote** are short and do not duplicate code - we simply call the constructor for the parent class and let that constructor deal with the **common attributes**. The **PianoNote** constructors need only worry about the attributes that are specific to piano notes. Similarly, the **destructor** for **PianoNote** doesn't need to do anything - because the compiler will call the **Note** destructor automatically after the **PianoNote** destructor has done its work, so as to ensure that proper cleanup of **common attributes** is performed.

Finally, and importantly, notice that **PianoNote** provides **its own implementation of get_sound_sample()**. This is possibly the most important bit since it is what allows **PianoNote** to **refine** or **specialize** its behaviour over that of a generic **Note**. This is called **method overriding** (do not confuse this with **method overloading** which we

discussed earlier) and is **explicitly indicated in the declaration of the function**.

When the **get_sound_sample()** function is called, the compiler checks if the **child class** has provided an **override**, and if so, it uses the **child class** function. If, on the other hand, the **child class** did not provide its own implementation, then the **parent's class** version is called.

This provides us with the flexibility to decide which functionality will be **refined** by the **child class**, and which functionality will be identical to what is provided by the **parent class**.

To complete the example, here's the declaration for **GuitarNote**:

```
class GuitarNote : public Note
{
    // GuitarNote is also a child of Note

    protected:
            int  stringNo;     // Which string was plucked

    public:
            GuitarNote():Note()
            {
                // Default constructor - it calls the default constructor of the parent class!
                this->stringNo=0;
            }

            GuitarNote(double t, double f, double d, int v, double strNo) : Note(t, f, d, v)
            {
                // Constructor with initialization data, calls the corresponding constructor
                // in Note for initializing common attributes.
                this->stringNo=strNo;
            }

            ~GuitarNote() override
            {
                // Nothing to do here, the compiler will call the destructor from
                // 'Note' after this function is called to ensure proper cleanup.
                // GuitarNote didn't add any dynamic data that needs cleanup!
                printf("** GuitarNote class destructor called!\n");
            }

            double get_sound_sample(double time_idx) override
            {
                // This is the implementation of the get_sound_sample
                // which should be different from a GuitarNote. Here
                // we will simply place a print statement to indicate
                // this function is being called

                printf("A GuitarNote with frequency %f is playing at time index %f\n",this->
                    frequency,time_idx);
                return 0;
            }

};
```

The definition for **GuitarNote** is similarly small, and doesn't duplicate any of the work that is already done in **Note**. Notice that we did not create a specialized version of **print_note_info()**, so if we call this function from a **GuitarNote**, we will in effect be using the function inherited from the **parent class**.

To complete the example, let's see how, once we have spent time and energy building a nice **class hierarchy**

of notes, we can easily write a program that is **easy to read and understand**, **doesn't have to deal with the complexities of different types of notes**, and yet **is able to use any of the types of notes we have provided**.

```
int main()
{
   // Let's declare an array of pointers to 'Note' objects
   // so we can simulate a loop that would 'play' music
   // from a set of notes.

   Note *all_notes[10];

   // Pay close attention, the array is for 'Note' objects!
   // but see what we can do now that we have a class hierarchy:

   all_notes[0]=new Note(1.25, 440, 2.0, 7);          // A generic 'Note'
   all_notes[1]=new PianoNote(1.5, 880, 1.0, 5,0,0);  // A 'PianoNote'
   all_notes[2]=new GuitarNote(2.0, 550, .5, 8,1);    // A 'GuitarNote'

   // We don't need to fill the remaining entries, but we could.
   // The key point here is: The program thinks its working with
   // 'Note' objects - but we can put *any derived* class objects
   // in this array. Why?
   // Because a PianoNote is a Note (just a specialized one!)
   // and a GuitarNote is also a Note
   // So software that works with 'Note' objects can use
   // any of the specialized versions without modification!

   // Let's see what happens when we want to 'play' sound from
   // these notes:

   printf("******************\n");
   printf("Playing all notes:\n");
   all_notes[0]->get_sound_sample(1.0);
   all_notes[1]->get_sound_sample(1.0);
   all_notes[2]->get_sound_sample(1.0);
   printf("\n");

   // Notice that the code above doesn't need to worry about the
   // fact each of these notes is actually a different type of
   // object! but the correct behaviour is obtained. For the
   // PianoNote, the PianoNote function is used, for the GuitarNote
   // the corresponding function is called.

   // Now see what happens when we call the print_note_info()
   // function:

   printf("******************\n");
   printf("Printing info for a generic Note:\n");
   all_notes[0]->print_note_info();
   printf("\nPrinting info for a PianoNote:\n");
   all_notes[1]->print_note_info();
   printf("\nPrinting info for a GuitarNote:\n");
   all_notes[2]->print_note_info();
   printf("\n");

   // The last call above is interesting. GuitarNote does not provide
   // a specialized version of 'print_note_info()', so the
   // function from 'Note' is automatically called instead!

   // That's it for this short example, let's clean up!
```

```
    printf("Deleting a generic Note\n");
    delete all_notes[0];
    printf("\nDeleting a PianoNote\n");
    delete all_notes[1];
    printf("\nDeleting a GuitarNote\n");
    delete all_notes[2];

    return 0;
}
```

The most important thing to notice in the listing above is that the program concerns itself only with **Note** objects - it doesn't know or care about **PianoNote**s or **GuitarNote**s (and if we had many, many more derived note classes, it wouldn't worry about those either). The program defines an array of **Note** objects, and then proceeds to use it to handle all of the types of notes that exist in our **class hierarchy**.

This works for a reason that is both elegant and intuitive: a **PianoNote** is a **Note**. A **GuitarNote** is also a **Note**. And if we had defined **ElectricGuitarNote** objects, we would realize that they are **GuitarNote**s and therefore also **Note**s. The result of this is that if we write a program that can handle **Note**s, the same program will be able to handle every derived class that is a specialized version of a **Note**. This is a fundamental idea in **OOP** and makes software design and implementation a lot easier and cleaner. Compare the code above with the long, cumbersome, and repetitive **get_sound_sample()** we tried to implement in **C**, and it should be clear that **OOP** has allowed us to create software that is significantly better in terms of the properties we described earlier in this Chapter.

Compiling and running the program in the listing above results in this output:

```
> ./a.out
******************
Playing all notes:
A plain Note with frequency 440.000000 is making sound at time 1.000000
A PianoNote with frequency 880.000000 is playing at time index 1.000000
A GuitarNote with frequency 550.000000 is playing at time index 1.000000

******************
Printing info for a generic Note:
This generic Note has frequency 440.000000
This generic Note has duration 2.000000
This generic Note has volume 7
This generic Note plays at time index 1.250000

Printing info for a PianoNote:
This PianoNote has frequency 880.000000
This PianoNote has duration 1.000000
This PianoNote has volume 5
This PianoNote's soft pedal is at 0.000000
This PianoNote's damper pedal is at 0.000000
This PianoNote plays at time index 1.500000

Printing info for a GuitarNote:
This generic Note has frequency 550.000000
This generic Note has duration 0.500000
This generic Note has volume 8
This generic Note plays at time index 2.000000

Deleting a generic Note
* Note class destructor called!
```

```
Deleting a PianoNote
** PianoNote class destructor called!
* Note class destructor called!

Deleting a GuitarNote
** GuitarNote class destructor called!
* Note class destructor called!
```

As you can see from the output - the program correctly handles each subtype of note - the compiler generates a program that automatically calls the correct function based on the subtype of note that is actually being used at any given time. We can see that the **overrides** work properly: The correct **get_sound_function()** is being called for each of the subtypes of note, and where **no override** is provided (as is the case for the **print_note_info()** in the **GuitarNote class**), the function provided by the **parent class** is called. Finally, the **destructors** are being called in order (as expected) - first the destructor for the **child class** and then the destructor for the **base class**.

In summary, **object oriented languages** provide a way for us to build rich hierarchies of related objects that provide a **conceptually clear model** for the **objects** our software will be working with, allow us to **implement functionality and store information** with **little or no duplication**, and with **flexibility** in deciding what behaviours will be refined by **derived classes**, and what behaviours will be inherited. A lot of tedious work is taken care of automatically (e.g. calling **constructors** and **destructors**, figuring out **which version** of a function to call), and this allows us to **think of**, **design**, and **implement** software at a more **abstract** level.

The result is better software - assuming that the proper amount of thought and care was put into the design of the **class hierarchy** and how to use the features of **OOP** to make the software solid with regard to our **good software wish list**.

This wraps up our very short look at **Object Oriented Programming** and **good software design**. This is really just a tiny overview of what is a complex, rich, and fascinating discipline. If you are interested in it, try a book on Software Design, or **OOP** next! There's just one more thing for us to discuss before the end of the Chapter.

6.9 Building programs that work - Part 6

Now that we have spent some time considering the issues involved in designing good software, and we know just how much work, thought, and time goes into building solid, useful libraries and software modules for others to use; it is time for us to think about the huge amount of software that has already been written and is available for us to build upon. The goal here is not to understand the **technical details** of how to use a **software module** implemented by someone else for our own projects. In this last section we want to learn about things we want to keep in mind when deciding whether or not we want to incorporate a specific software **module** developed by someone else into our own project. As we will see, the choices we make in this regard can have some meaningful impact for our project, including imposing certain conditions regarding how our work can be used, distributed, and/or commercialized.

The most important thing to remember here is that **we can not simply take software we did not create and put it into our project**. We have to think through how to properly **give credit** where and as needed, and we have to be certain we understand any **copyright, licensing, and distribution** conditions that come with any software we pick up to use with our projects.

6.9.1 Common libraries distributed with the compiler and operating system

The first source of software that we can definitely use for our own projects consists of all the **system libraries** that are distributed with your compiler. They include a fairly wide range of common functionality that many projects will need. You've already used some of these libraries: **stdio**, **stdlib**, **string** and **math** at the very least. These are provided to us free of any charge and with no expectation regarding how we will distribute or commercialize our programs.

If you want to see which libraries are included with your compiler, you want to find the directory that contains the **header** files (on a Linux system this is typically in **/usr/include**, there will be a corresponding location on Windows and Mac).

Beyond the libraries already included with your compiler, there are many libraries you can in install to provide specialized functionality, and this includes anything from **handling image files**, **network connectivity**, **machine learning**, **security and cryptography** and much more. Compiler libraries are usually distributed under the **GNU Lesser General Public License** (`https://www.gnu.org/licenses/lgpl-3.0.en.html`). This is something we should think about for a moment - it means there are **terms and conditions** that we have to be **aware of** and **willing to accept** if we use any of these libraries within our programs.

Fortunately, the **LGPL** is fairly permissive - this makes sense, if these libraries placed too many conditions on what users can do with programs that contain them, no one would use them. An important section of the license states the following: **You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications**. This tells us that if we incorporate libraries distributed under the **LGPL** within our programs, we are not restricted in how we can distribute our work and what licensing terms we apply to the resulting programs as long as the users of the software are still able to freely modify and reverse engineer **the parts of the libraries** that were used within the software.

In other words - we can't **lock down** any portion of the libraries distributed under **LGPL** that are part of our software, which is not usually a problem unless we are **distributing** the library's source code in some form. Most commercial software comes in the form of **executable programs** that do not contain the source code for libraries used to build them, so there are no restrictions placed by the **LGPL** on such software and how we can distribute or commercialize it.

But, you should always check the licensing terms for any **non-standard libraries** that you may be thinking of using and make sure you are aware of what restrictions, if any, they place on software you are building.

6.9.2 Free and Open Source Software (FOSS)

An immense variety of software out there is distributed as **Free and Open Source Software (FOSS)**. You can find **FOSS** for pretty much any purpose and written in any programming language. The part that makes **FOSS** interesting to us is that **modules** distributed under **open source** licenses allow us to **use, modify, and distribute** the **source code** for the module **free of charge**. That means we have access to an incredibly rich software base which could allow us to build fairly sophisticated programs quickly and with more functionality than we could provide if we had to write every piece of the software ourselves.

However, before we go ahead and start using that wonderful package we just saw on **GitHub**, we should

consider:

- There are many different licenses for **FOSS**. The type of license makes a big difference on what you can do with the resulting software.
- **Permissive** licenses such as **BSD, MIT, or Apache** allow you to distribute and sell resulting software with almost no restriction.
- **CopyLeft** licenses such as **GPL** require that the resulting software's **source code** be provided to users - in other words, any software you write that uses **modules** distributed under such a license **must remain open source**.

So depending on what we intend to do with the software we are developing, we may or may not want to use a particular **FOSS module**. Our process should always include **checking the license terms** for any piece of software we want to use as part of our code. And we have to be **aware of the terms** of the **license** and **comfortable complying with any restrictions it places on our software**.

One more thought along these lines: If we develop a software package and we want to distribute it as a **FOSS** project, we definitely want to give considerable thought to the issue of choosing **which license** we want to be applied to our software. The choice will place conditions on what other developers can do with it, and as a result it will also change the set of projects that will ultimately feature our work. So becoming familiar with these licenses will eventually be important.

Note

Here is another important thing to remember: whether you are using standard **C** libraries, or whether you are using **FOSS** modules, keep the following in mind:

- We can not assume the software is **correct** (free of **bugs**) simply because it is used extensively by many developers.
- We can not assume the software is **safe** in terms of **data privacy** or **prevention of software vulnerabilities**.
- We can not assume the software is **efficient** in its use of resources.

All of these issues **have to be considered** while deciding whether or not to incorporate a particular software **module** into something we are building. However, there are also advantages to using **open source** software:

- Because it is **open source**, there are no **hidden** features - it is difficult to hide **malware** and **spyware** on software that can be openly inspected by anyone at anytime.
- While we can't assume the software is **bug free**, known problems are **openly documented** and there is often a solid process for **resolving bugs** and **updating the software** - this can be slow, but at least developers are aware of known problems.
- We can benefit from the experience of a large group of **software developers** who are using or have used the software and can provide help if issues arise.

As long as we are aware of what is involved in using **open source** software, we can always find a way to get the most out of the work many others have done and made available for the benefit of everyone.

6.9.3 Large Language Models and other A.I. tools

A completely different source of help in building software is now extensively available in the form of a number of **Artificial Intelligence** platforms whose functionality and ability to interact with users has grown incredibly fast in a very short span of time. Of particular note among current A.I. tools are the so-called **Large Language Models (LLMs)** which power platforms such as **ChatGPT** and **Gemini**.

These tools do anything from **providing help with concepts, and explaining details about algorithms**, to **summarizing algorithms in pseudocode**, to **providing program code that performs a specified task** and **suggesting tests** for software we are writing. Current **LLMs** are already incredibly powerful, and are becoming better and more capable by the day. **We should definitely learn how to use them** to support our work and increase our ability to build good software.

But we have to do this with full awareness that we are **entirely responsible** for the result of the work we produce regardless of whatever information or help we obtained from A.I. tools. And we should remember that **the information we obtain is not always correct**. This includes both **conceptual** information, as well as **program code**. Let's see a couple of examples of this to make the point clear.

Suppose we want to consult **factual information** about a specific **programming language** because we want to understand what its features are, and what we can expect from it. This type of information is what we may expect to find in a **book** or **technical blog**, or perhaps **class lecture notes**, and as such it is **often reliably and accurately** provided by the latest **LLMs**. Fig. 6.11 shows an example interaction in which we are attempting to find out how well **Python** implements the key components of **OOP** we discussed earlier in the chapter.

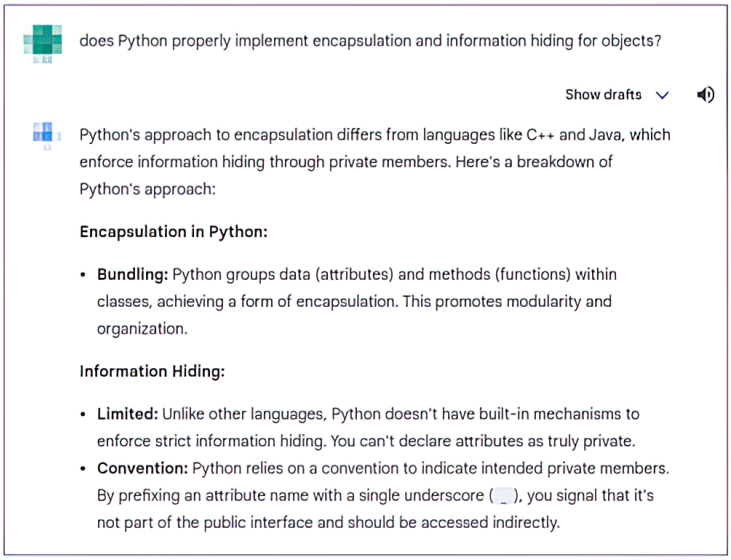

Figure 6.11: Asking an **LLM** how well **Python** supports **encapsulation** and **information hiding**.

As you can see, the response is correct and describes concisely what **Python** does in terms of **encapsulation** and **information hiding**. The response obtained was actually longer and provided examples supporting each of the claims the **LLM** made. Current **LLMs** are quite good at summarizing information from multiple sources and presenting it in a concise form, which can be very useful when we are trying **to verify our understanding** of a concept, algorithm, or problem. The flipside of this is that **the more specific the query, the higher the likelihood the information may be not entirely correct** - as an example, suppose we are trying to recall what the **complexity** of a particular **graph operation** is, and we decide to ask an **LLM** about it. The resulting conversation is shown in Fig. 6.12

Notice two things: Firstly **the LLM will provide a very confident answer that tends to seem correct** and therein the danger - if you are **not constantly thinking through what you're being told**, and making sure it is consistent with **your own hard-earned understanding** of how things work in computer science, you could easily just **take the explanation as correct and go with it**. Which would be a mistake. In this case, we know (from what we learned in Chapter 5) that **this answer is not correct**, and prodding the **LLM** results in confirmation of our own understanding of things.

Why does this happen? - what we must keep in mind is that **the current generation of LLMs** works by **predicting each successive token** (which is a word, or symbol) in the answer **given the prompt and any part of the answer that has already been produced**. It is a matter of **choosing among all possible tokens** the one that has the **highest likelihood** of being the correct one. As it turns out, with a large enough **LLM**, and with a sufficiently large and rich **set of training data**, the **LLM** can do this task extremely well. Of course, this is an **over simplification** and truly understanding how the **LLM** works requires long and serious study.

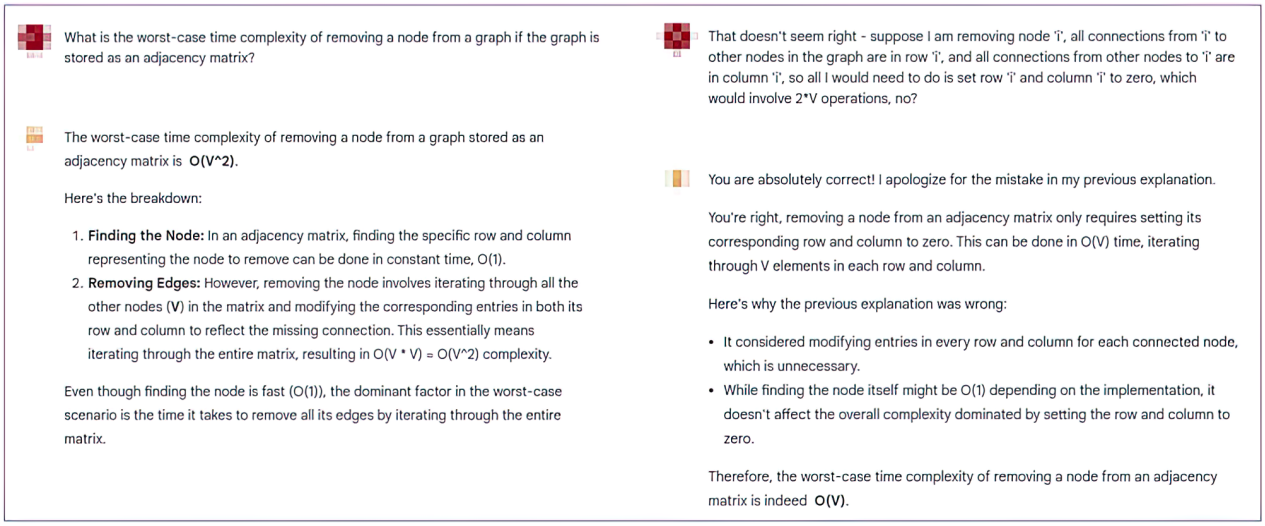

Figure 6.12: A more **specific technical query** does not produce the right result.

But it is enough to tell us something: The **LLM** is not **checking its own answer** against a **thought process** of some sort, or **checking it is accurate** against its own **knowledge base**. It just predicts tokens one after the other and comes up with an answer that has high probability of being correct. You should expect that for material for which there is **a lot of available training information** - such as **general concepts** in computer science, or implementations of **common algorithms**, or material that is discussed in books, blogs, lecture notes, and so on; the **LLM** will likely come up with the correct response - in effect, it has seen the answer (many times) before.

For more specialized queries, ones for which the **training data** may not contain enough examples (or in fact may even contain incorrect information), the **LLM** should be expected to have a harder time and can possibly give us an incorrect answer, it will do this with great confidence, and it will **fail to notice obvious mistakes** - to drive home this point, see the conversation in Fig. 6.13 in which a fairly obvious mistake is made for a technical question that a human observer would not be confused by.

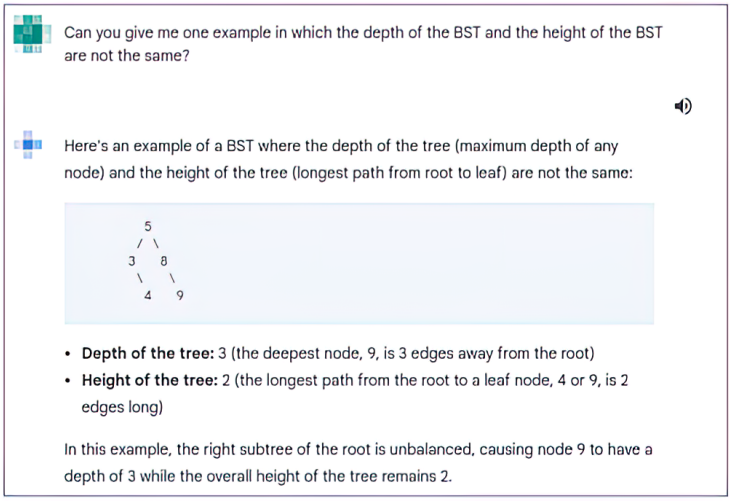

Figure 6.13: The answer by the **LLM** is clearly incorrect - inspection of its own **BST** example shows there are only **2 edges** between **5** and **9**, yet the **LLM** confidently asserts that there are **3 edges** between these nodes.

The point here is **not to criticize** or **make less of** the usefulness of **LLMs** - they are fantastic tools and **used correctly** they can make our work a lot **easier**, more **fun**, and **improve your productivity significantly**. But the key to this is that we must always remember **to think through** the information we are getting back, consider whether or not it is **consistent with what we have learned** and **what we would expect**, as well as **common sense**, and then decide **how to use the information**. If anything seems **odd** or **not consistent** with what we think is correct - then we must do the work of **asking further questions** and/or **checking with other sources** before we proceed with information whose accuracy we can't determine by ourselves.

A special problem is posed by the use of advanced A.I. tools for **generating code**. Much like with conceptual answers, **LLMs** can be expected to provide **solid, correct, working** code for **very common problems**. For instance, if you need code to **insert a node into a linked list** - well, chances are that the **training data** for coding A.I. contains hundreds if not thousands of examples of this procedure, in different languages, with different types of data structures. So the answer you get will likely be correct.

So A.I. tools for code generation can be incredibly useful, once more, **if used correctly**. In particular, they can be very helpful with tasks that require us to write in a language we are not familiar with. For instance, Fig. 6.14 shows the result of requesting a fairly specialized Linux shell script to carry out a file counting task.

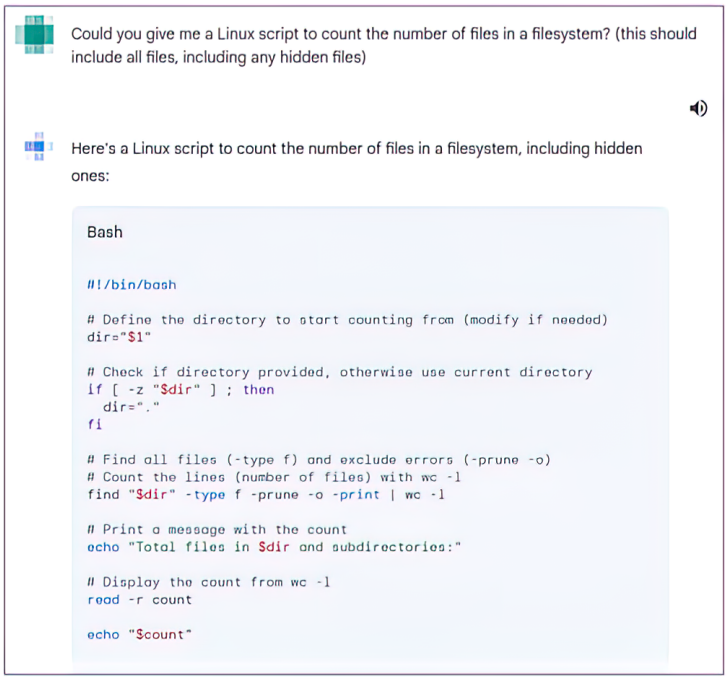

Figure 6.14: Current **LLMs** that can produce program code have become quite capable, and can produce fairly specialized, correct, and well documented code for tasks they have encountered in their training set (or those that are very similar).

The resulting script works, and the **LLM** output also included instructions on how to use it, examples, and comments on possible variations or different ways to carry out this particular task. In all, it is **incredibly helpful** and **saved a lot of time**. But of course, just as with factual answers **the program code you obtain may be incorrect** and the **LLM** will **not notice it**.

There is also a **potential issue with copyright** - because these tools generate program code based on the

examples it has seen in **training data**, and the training data consists of programs that have been written by human developers, the code produced by the **LLM** can often **closely resemble** or even **replicate** existing programs or parts of programs written by developers who put their code somewhere accessible online.

Because it is often the case that **LLMs** use code from online repositories as **training data**, often **without the knowledge or explicit consent** from the authors, and **offering no financial compensation** for the use of their code. It is **at the present not entirely clear** that program code produced by an **LLM** is free of potential **copyright infringement** issues. At the very least, we as users of an A.I. tool that produces code we intend to use have to be **fully aware** that we are potentially taking advantage of someone else's work - without due credit being given, and providing nothing in return. Not an ideal situation.

So, how are we to proceed with code-generating tools? The rules are changing quickly, the **LLMs** themselves are evolving very fast and their capabilities are improving and expanding in a very short time frame. But at the present time, here are a couple of **generally good ideas** to keep in mind if you intend to use **LLMs** to generate code for you:

- We must **always** carefully **read through**, the code we obtain, **carefully following the process and understanding the algorithm and how it is implemented** to determine whether is does the right thing. If **we do not understand** either the **algorithm**, or **the implementation**, or we **can't decide whether it is correct or not**, then **we should NOT use it**.
- We must be **aware** of potential **copyright issues** and check whether the use we are making of the automatically generated code may turn into a problem. If working in industry we must **always check the company's policy on use of A.I. tools for generating code** and follow their guidelines. If we are going through a program of study, for instance at a University, we must be aware that many institutions forbid the use of A.I. tools or place very strong restrictions on how and where they can be used - we should **check the rules that apply** and steer clear of potential trouble.
- Always **develop a thorough testing framework** - just as we would for code we are developing ourselves, for testing any A.I. generated code. The point is that since the code we obtain is the result of training data which is itself human-generated code, there is **every expectation** that it **can and will contain bugs**. So we have to **thoroughly test it** and **check it for correctness** as discussed in Chapter 3.

We can always use an **LLM** to improve our own understanding of things, and automated code generation can be a powerful learning tool. So we should consider **getting help with specific syntax or use cases in a language we are learning**, or **request examples of how a particular algorithm works**, or **asking for examples on how to work with a particular tool, framework, or API**. There are many, many possible uses of an **LLM** in the context of computer science and software development that make the most of the **LLMs** capabilities without tripping on issues such as we described above. So it is worth spending time learning how to interact with these tools, and training ourselves to correctly interpret, learn from, and when necessary correct the information we receive from them.

With that, we wrap up this book on introductory computer science, and on how to build programs that work. Now let's get out there, apply all that we have learned to the task of solving exciting problems, and choose what we want to learn next. This book is just a glance at a fascinating and incredibly rich area of science, and whatever our interests and goals, there will be more to explore that will match our interests and help us get to where we are going.