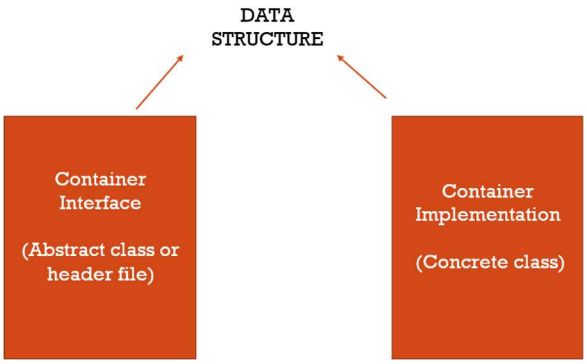
Separate Compilation in C++

C++ header files serve the purpose of providing *function prototypes* that separate function definitions from their implementation and can be deemed an *interface* for the class.[1]

C++ classes are (most commonly) split into two files. The *header file* has the extension of .h and contains class definitions and functions. The implementation of the class goes into the .cpp file. Moreover, .cpp files should not (ever) be used in an include statement in files that need to utilize a class. Rather, include the .h file instead.

A header file informs *what* the class can contain/do, and the .cpp file then fulfills *how* its done. We can visualize it this way:



A makefile, and most IDEs, will only recompile the files that have changed. The header, once defined, is seldom modified; when a class definition and class implementation are split up this way, files that include the header don't need to be recompiled (changes to class implementation files are managed by the linker). This may not seem like a big deal, but in large projects it has more of an impact.

Example

File**: Bin.h**

```
class Bin
{
 private:
    int num;
 public:
    Bin();
    Bin(int n);
    int getNum();
};
```

File**: Bin.cpp**

```
#include "Bin.h"

Bin::Bin() : num(0) { }
Bin::Bin(int n): num(n) {}
int Bin::getNum() {
   return num;
}
```

File**: main.cpp**

```
#include <iostream>
#include "Bin.h"
using namespace std;

int main() {
    Bin  b(210);
    cout << b.getNum() << endl;
    return 0;
}
```

To compile this from the command line we would use:

```
g++ main.cpp Bin.cpp
```

[1] Java has the *interface* definition for abstraction. By implementing an interface in Java, functionality is ensured.

Using preprocessor conditional compilation

Header files may be required in multiple files, yet C++ doesn't allow redefining a class. To prevent redefinition, we utilize preprocessor directives `#if`, `#else`, `#elif`, `#ifdef`, `#ifndef` and `#endif`.

Our previous Example 1 compiled main.cpp and Bin.cpp separately, so the inclusion of Bin.h in each file didn't cause a problem. The next example shows a redefinition problem where two classes in main use Bin.

Example 2.

File: Fin.h

```
#include "Bin.h"
class Fin
{
 public:
   Bin n;
};
// No Fin.cpp is needed
// as there's nothing to
// implement.
```

```
#include <iostream>
#include "Bin.h"
#include "Fin.h"
using namespace std;
int main()
{
  Bin b(210);
  cout << b.getNum() << endl;
  Fin f;
  cout << f.n.getNum() << endl;
  return 0;
}
```

Now if we try to compile:

g++ main.cpp Bin.cpp

```
In file included from Fin.h:1:0,
                 from main.cpp:4:
Bin.h:2:7: error: redefinition of 'class Bin'
 class Bin
       ^
In file included from main.cpp:3:0:
Bin.h:2:7: error: previous definition of 'class Bin'
 class Bin
       ^
```

To fix the compile error, we can use `#ifndef`[2]. This is a *directive* as it is a message to the compiler. It tells the compiler to ignore what follows if it has already seen this defined name before. The format looks like this:

```
#ifndef BIN_H
#define BIN_H
 <class definition … >
#endif
```

If `BIN_H` has not yet been defined, it gets defined, and the header code is processed. If `BIN_H` has already been defined, the processor skips over the definition.

A good practice is to always use `#ifndef` in header files and to use a name related to the class.

[2] Alternately, use `#pragma once` at the top of the file. This is Visual Studio's default behavior.

Separate Compilation

So far, we split the header from the implementation. We're still compiling both all the time when we run the g++ command. To really get separate compilation we need to:

1. Compile each .cpp file into an object file, which contains machine code for that file
2. Link each object file into an executable

To compile into an object file, use the command flag of –c:

g++ -c main.cpp Bin.cpp

This produces a file `main.o` and `Bin.o`. We can then link them. We can use g++ again to do this:

g++ main.o Bin.o

We now have the default executable `a.out` program which we can run. An efficiency step is possible here because if `Bin` never changes but `main` does change, then compile just `main` via:

g++ -c main.cpp

Then we can link them once more without the overhead of compiling `Bin.cpp`:

g++ main.o Bin.o

For really large programs this can be a significant savings in compile time. An IDE automatically does the compiling and linking for you, which is one reason they are so useful and popular.

The make utility

It can be a pain to compile lots of files. Once again, this functionality is provided for you as part of an IDE. If you are compiling from the command line, you can use the **make** utility. If you run the command "make" then this program will look for a file named "makefile" (or "Makefile") that contains information about program dependencies and what needs to be compiled. Here is an example:

```
CFLAGS = -O
CC = g++
BinTest: main.o Bin.o
	$(CC) $(CFLAGS) -o BinTest main.o Bin.o
main.o: main.cpp
	$(CC) $(CFLAGS) -c main.cpp
Bin.o: Bin.cpp
	$(CC) $(CFLAGS) -c Bin.cpp
clean:
	rm -f core *.o
```

This says that `BinTest` depends on `main.o` and `Bin.o` and the second line says how to compile it. In turn, `main.o` is created by compiling `main.cpp` with the –c flag. We can compile by typing:

make

or

make BinTest

The *targets* are compiled and linked to produce an executable named `BinTest` (this is identical to `a.out` with a better name). Console output from typing `make`:

```
g++ -O -c main.cpp
g++ -O -c Bin.cpp
g++ -O -o BinTest main.o Bin.o
```

Only files that have changed are recompiled when we type “make” again. We can use the target of “clean” to delete object files and core dumps. This is often done to make sure the latest version of source files is used.