Modularity

Modularity is about breaking a program up into small, re-usable pieces rather than a few large, monolithic blocks. A program written in a modular style has many advantages over one that isn't. Despite being broken into smaller pieces, a modular program tries to avoid duplicated code, resulting in a smaller total amount of code. Reducing or eliminating duplicated code also makes code easier to test and debug, because each piece of functionality is only written once — it can only be buggy in that one place, and fixing bugs in that place fixes them for the entire program. Modular code makes later modifications to the program easier as well. Since different pieces of functionality are implemented in different sections of code, it is easy to add more functionality, or use the existing code to implement another program with similar functionality.

We will look at a small program that has not been written in a modular fashion and investigate how it could be redesigned in a modular way. We will then split the program into two separate programs, with the shared functionality being implemented in a shared file.

A non-modular program

The program we will look at generates the given term in a mathematical sequence. The two sequences it supports are fibonacci numbers, and prime numbers. The maths used to generate these are not important to us here — all that matters is that generating the sequences follows the same pattern, as we will see.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <stdbool.h>
5
6  long calculatePrime(int n) {
7      long *list = malloc(sizeof(long) * n);
8      for (int i = 0; i < n; i++) {
9          if (i == 0) {
10             list[i] = 2;
11         } else {
12             long possiblePrime = list[i-1];
13             bool isPrime;
14             do {
15                 possiblePrime++;
16                 isPrime = true;
17
18                 for (int j = 0; j < i; j++) {
19                     if (possiblePrime % list[j] == 0) {
20                         isPrime = false;
21                         break;
22                     }
23                 }
24             } while (!isPrime);
25             list[i] = possiblePrime;
26         }
27     }
28     long result = list[n - 1];
29     free(list);
30     return result;
31 }
32
33 long calculateFibonacci(int n) {
34     long *list = malloc(sizeof(long) * n);
35     for (int i = 0; i < n; i++) {
36         if (i <= 1) {
```

```
            list[i] = i;
        } else {
            list[i] = list[i - 1] + list[i - 2];
        }
    }
    long result = list[n - 1];
    free(list);
    return result;
}

int main(int argc, char *argv[]) {
    int n = strtol(argv[2], NULL, 10);
    if (strcmp(argv[1], "fibonacci") == 0) {
        printf("%ld\n", calculateFibonacci(n));
    } else if (strcmp(argv[1], "prime") == 0) {
        printf("%ld\n", calculatePrime(n));
    }
    return 0;
}
```

This program takes two command-line arguments: the name of the sequence to use (either "prime" or "fibonacci") and which element to generate. Its two important functions are `calculatePrime` and `calculateFibonacci`. These functions calculate the n-th term in a sequence by calculating the entire sequence up to that point, then returning the last term calculated. So while the maths being used to generate each element is different, there is duplicated functionality here — the only place these functions differ is calculating the individual elements from the values of the previous ones.

Adding modularity

We would like to factor out the repeated functionality described above. We will use function pointers to do this. To generate the next value in the sequence, the previous values are needed, as is the position in the sequence that we are up to (how many elements have we calculated already). We can represent this with a function:

```
long generateNextElement(long *previous, int n);
```

We can write a function with this type to generate the next prime number, and one to generate the next fibonacci number. If we remove this functionality (generating the next element) from each `calculateFibonacci` and `calculatePrime` we are left with the same function, which doesn't know how to calculate each element. We can add another argument to this function, which is a function pointer of the type given above.

```
long calculateNthOfSequence(long (*generator)(long *, int), int n);
```

This means that the calling code can specify which element generation function to use when it calls the function, and if we want to add a type of sequence we only need to write a new element generation function and not repeat the sequence creation code. The whole program is given below:

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

long calculateNthOfSequence(long (*generator)(long *, int), int n) {
    long *list = malloc(sizeof(long) * n);
    for (int i = 0; i < n; i++) {
        list[i] = generator(list, i);
    }
    long result = list[n - 1];
```

```
12      free(list);
13      return result;
14  }
15
16  long nextPrime(long *previous, int number) {
17      if (number == 0) {
18          return 2;
19      } else {
20          long possiblePrime = previous[number - 1];
21          bool isPrime;
22          do {
23              possiblePrime++;
24              isPrime = true;
25              for (int i = 0; i < number; i++) {
26                  if (possiblePrime % previous[i] == 0) {
27                      isPrime = false;
28                      break;
29                  }
30              }
31          } while (!isPrime);
32          return possiblePrime;
33      }
34  }
35
36  long nextFibonacci(long *previous, int number) {
37      if (number <= 1) {
38          return number;
39      } else {
40          return previous[number - 1] + previous[number - 2];
41      }
42  }
43
44  int main(int argc, char *argv[]) {
45      int number = strtol(argv[2], NULL, 10);
46      if (strcmp(argv[1], "fibonacci") == 0) {
47          printf("%ld\n", calculateNthOfSequence(nextFibonacci, number));
48      } else if (strcmp(argv[1], "prime") == 0) {
49          printf("%ld\n", calculateNthOfSequence(nextPrime, number));
50      }
51      return 0;
52  }
```

Multiple soure files

A common pattern in software projects is having multiple programs that both require a certain piece of functionality. To mimic this situation, we are going to split the above program into two separate programs, one for calculating fibonacci numbers and one for calculating primes. The fibonacci program will need `nextFibonacci` and the prime program will need `nextPrime`. but both will need `calculateNthOfSequence`. Copying `calculateNthOfSequence` into both programs would defeat the purpose of making the program modular, so the only solution is to have a library file which both programs refer to.

We will end up with the following files:

- `sequence.c` and `sequence.h` for the shared functionality.
- `prime.c` for the prime generation program.
- `fibonacci.c` for the fibonacci generation program.

The `sequence` files contain any functions and types that both programs require access to. Any functions called directly by `prime.c` and `fibonacci.c` should have prototypes in the header file (`sequence.h`) and implementations in `sequence.c`. If a function is called by another function in the sequence files but no others, it does not need a prototype in the header — only "publicly visible" things belong in the header. Similarly, any type (struct, enum, etc.) that is referred to by the main files should be in the header.

`sequence.h`:

```
1  #ifndef SEQUENCE_H
2  #define SEQUENCE_H
3
4  int calculateNthOfSequence(long (*generator)(long *, int), int n);
5
6  #endif
```

`sequence.c`:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include "sequence.h"
5
6  long calculateNthOfSequence(long (*generator)(long *, int), int n) {
7      long *list = malloc(sizeof(long) * n);
8      for (int i = 0; i < n; i++) {
9          list[i] = generator(list, i);
10     }
11     long result = list[n - 1];
12     free(list);
13     return result;
14 }
```

There are two extra things to note here. Firstly, `sequence.c` includes `sequence.h` so that we don't need function prototypes for functions declared in the header, and we can use any types from the header. The second thing to notice is the preprocessor lines surrounding the contents of the header file. These are known as "include guards". Since `#include` simply places the contents of a file into another one, large projects can end up with a situation where a given header in copied multiple times into a single file, causing compilation errors. The include guards prevent this from happening. When making your own headers, make sure that each header uses a unique constant like `SEQUENCE_H`. All that is left are the two program files:

`fibonacci.c`:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "sequence.h"
4
5  long nextFibonacci(long *previous, int number) {
6      if (number <= 1) {
7          return number;
8      } else {
9          return previous[number - 1] + previous[number - 2];
10     }
11 }
12
13 int main(int argc, char *argv[]) {
```

```
    int n = strtol(argv[1], NULL, 10);
    printf("%d\n", calculateNthOfSequence(nextFibonacci, n));
    return 0;
}
```

prime.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "sequence.h"

long nextPrime(long *previous, int number) {
    if (number == 0) {
        return 2;
    } else {
        long possiblePrime = previous[number - 1];

        bool isPrime;

        do {
            possiblePrime++;
            isPrime = true;

            for (int i = 0; i < number; i++) {
                if (possiblePrime % previous[i] == 0) {
                    isPrime = false;
                    break;
                }
            }
        } while (!isPrime);

        return possiblePrime;
    }
}

int main(int argc, char *argv[]) {
    int n = strtol(argv[1], NULL, 10);
    printf("%d\n", calculateNthOfSequence(nextPrime, n));
    return 0;
}
```