Beispiel `primes`

Dr. Günter Kolousek

28. Juli 2016

1 Allgemeines

- Drucke dieses Dokument **nicht** aus!
- Halte **unbedingt** die Coding Conventions ein! Zu finden am edvossh!
- Lege für dieses Beispiel wieder ein entsprechendes Unterverzeichnis mit Beispielnummer und Beispielname an!
- Verwende den angegebenen Texteditor!

2 Aufgabenstellung

Schreibe ein `C++` Programm `primes`, das alle Primzahlen kleiner gleich 100 ermittelt und ausgibt.

3 Anleitung

1. Ganz am Anfang dieses Beispiels wollen wir uns gleich einmal um die Struktur unseres Miniprojektes im Dateisystembaum kümmern. Im letzten Beispiel haben wir uns schon mit CMake befasst, das uns das Builden des Projektes sehr erleichtert. Aber alle von CMake generierten Dateien sind im gleichen Verezeichnis wie unsere `C++` Dateien. Das ist gar nicht schön.

   Lege deshalb gleich einmal im Beispielverzeichnis das zusätzliche Verzeichnis `build` an, das in weiterer Folge alle von CMake und von `make` erzeugten Dateien erhalten soll.

   Dazu benötigen wir natürlich wieder eine Datei `CMakeLists.txt`, die jetzt einfach nur die folgenden Zahlen enthalten soll:

```
cmake_minimum_required(VERSION 3.5)
project(primes)
add_executable(primes main.cpp primes.cpp)
```

Soweit hat sich ja einmal nichts geändert. Erstelle jetzt ein leeres Modul `primes` und die leere Datei `main.cpp` und dann weiter zum nächsten Punkt.

2. Bevor wir uns mit dem eigentlichen Finden aller Primzahlen befassen, wiederholen wir kurz um was es sich bei einer Primzahl handelt. Vielleicht als Abwechslung einmal eine Definition gefällig?

   > Eine natürliche Zahl n heißt **prim**, wenn $n \geq 2$ ist und es keine natürlichen Zahlen a und b mit $a > 1$ und $b > 1$ gibt, sodass $n = a \cdot b$ ist.

   Fein, jetzt wissen wir was eine Primzahl ist, aber wie können wir überprüfen, ob es sich bei einer gegebene Zahl um eine Primzahl handelt?

   Ein einfacher (aber nicht sehr guter) Ansatz ergibt sich direkt aus der obigen Definition. Wenn $n \geq 2$ ist, dann alle natürlichen Zahlen a und b ausprobieren mit $1 < a, b \leq n/2$ und überprüfen, ob $n = a \cdot b$ ist.

   Wie kann man dies direkt in `C++` umsetzen? Schreibe eine Funktion `bool is_prime0(unsigned int n)`, die diesen Algorithmus genau in der Form dieser Definition implementiert (auch wenn dies nicht sonderlich klug ist). Beachte weiters , dass du dafür zwei in einander verschachtelte Schleifen benötigst.

   Diese Funktion kommt in das Modul `primes` und das Testen kommt wieder in die Datei `main.cpp`. Denke daran, dass die Ausgabe eines boolschen Wertes in `C++` standardmäßig als 0 bzw. 1 (für `false` bzw. `true`) ausgegeben wird (wir kümmern uns gleich im nächsten Punkt darum). Bzgl. Testen schlage ich vor, dass du dir für alle natürlichen Zahlen bis inklusive 13 das Ergebnis ausgeben lässt.

   Der entscheidende Unterschied bezüglich dem Übersetzen ist, dass du in das Verzeichnis `build` wechselst und dort die folgenden beiden Befehle eingibst:

   ```
   cmake ..
   make
   ```

   Dann werden alle von `cmake` und `make` erzeugten Dateien im Verzeichnis `build` angelegt werden und unsere `C++` Dateien liegen schön übersichtlich im Projektverzeichnis.

3. Wir haben zwar einen funktionsfähigen Algorithmus, aber ein offensichtliches Manko liegt darin wie ein `bool`-Wert in einem Ausdruck der Form `cout << b` ausgegeben wird. An sich ist das ja in Ordnung. Wir wissen, dass `C++` intern einen `false` Wert als 0 und einen `true` Wert als 1 behandelt.

   Weiters wissen wir, dass eine ganze Zahl in einem booleschen Ausdruck genau dann als `false` aufgefasst wird, wenn die Zahl den Wert 0 hat, anderenfalls als `true`.

Will man anstatt 0 bzw. 1 lieber `false` bzw. `true` in der Ausgabe, dann kann man das auf folgende Art und Weise erreichen:

```
cout << boolalpha << false << " : " << true << endl;
cout << noboolalpha << false << " : " << true << endl;
```

Mit diesem Code wird die Ausgabe folgendermaßen aussehen:

```
false : true
0 : 1
```

Bei `boolalpha` bzw. `noboolalpha` handelt es sich um sogenannte Manipulatoren (die eben den Stream manipulieren) und sich im Namensraum `std` befinden. Durch Inkludieren von `iostream` stehen diese dann zur Verfügung.

Zu beachten und aus dem Beispiel auch zu erkennen ist, dass diese den Stream solange manipulieren bis ein gegenteiliger Manipulator die Ausgabe wiederum umschaltet.

4. So, wenn das jetzt funktioniert, dann wollen wir uns wieder unserer Projektstruktur zuwenden und die nächste Verbesserung in unserem Projekt vornehmen, indem wir die `C++` Dateien in ein eigenes Unterverzeichnis `src` verschieben. Also, `src` anlegen und `C++` Dateien hinein.

   Unser Projektverzeichnis wird jetzt folgende Struktur aufweisen:

   ```
   07_primes
     CMakeLists.txt
     src
       main.cpp
       primes.cpp
       primes.h
     build
   ```

   Jetzt wird das Projekt allerdings nicht mehr übersetzen, da `cmake` die `C++` Dateien nicht mehr finden wird. Deshalb müssen wir die `CMakeLists.txt` abändern:

   ```
   cmake_minimum_required(VERSION 3.5)
   project(primes)
   add_executable(primes src/main.cpp src/primes.cpp)
   ```

   Vergiss nicht `cmake` erneut aufzurufen, damit `cmake` die Datei `CMakeLists.txt` erneut einliest und die geänderte Konfiguration erkennt.

5. Das ist ja einmal gar nicht so schlecht, aber zwei Dinge stören noch:

   - Es ist lästig jede neue `.cpp` Datei in die `add_executable` Direktive mit aufzunehmen.

- Die Headerdateien stellen die Schnittstelle zu unseren Modulen dar und sollen anderen Benutzern zugänglich gemacht werden. Es ist allerdings nicht notwendig, dass die anderen Programmierer die Implementierung des Moduls sehen. Manchmal ist es sogar explizit unerwünscht, wenn man nämlich eine Bibliothek verkaufen will.

Um diese beiden Probleme zu lösen gehen wir folgendermaßen vor:

a) Wir legen ein neues Verzeichnis `include` an (im Projektverzeichnis auf der gleichen Ebene wie `src` oder `build`) und verschieben dorthin `primes.h`.

b) Wir ändern unsere `CMakeLists.txt` jetzt folgendermaßen ab:

```
cmake_minimum_required(VERSION 3.5)
project(primes)

include_directories(include)

file(GLOB SOURCES "src/*.cpp")

add_executable(primes ${SOURCES})
```

Die Direktive `include_directories` gibt ein Include-Verzeichnis bekannt und die `file` Direktive speichert in die Variable `SOURCES` alle Dateien, im `src` Verzeichnis, die die Endung `.cpp` haben (als eine Liste). `${SOURCES}` greift auf den Wert der Variable `SOURCES` zu und fügt diese an ihrer statt ein.

6. Wenn du den Algorithmus in der folgenden Form gemäß der Definition implementiert hast, dann sieht diese Implementierung kodiert in Python in etwa folgendermaßen aus:

```
res = True
if n < 2: return False;
for a in range(2, n // 2):
    for b in range(2, n // 2):
        if n == a * b:
            res = False
return res
```

Das entspricht zwar der Definition, die angibt, dass für *alle* a und alle b im entsprechenden Intervall die Bedingung zu überprüfen ist. Allerdings ist dies nicht sehr klug, da es keinen vernünftigen Grund gibt die Schleifen weiter abzuarbeiten wenn es sich herausgestellt hat, dass es sich um keine Primzahl handelt.

Ändere deshalb deine Implementierung entsprechend ab!

7. Auch mit dieser Verbesserung ist dieser Algorithmus auf Grund der beiden in einander verschachtelten Schleifen nicht effizient.

Wie kann eine grundlegende Verbesserung erreicht werden? Dazu versuchen wir das Problem besser zu verstehen…

Eine Zahl ist dann nicht prim, wenn diese in zwei Faktoren aufgespalten werden kann. Eine der beiden Faktoren muss kleiner oder gleich der Wurzel dieser Zahl sein. Wieso? Nehmen wir einmal an, dass für eine Zahl eine ganzzahlige Wurzel existiert, wie z.B. $16 = 4 \cdot 4$. Dann kann jede weitere Darstellung, wie z.B. $16 = 2 \cdot 8$ nur so aussehen, dass der erste Faktor kleiner als die Wurzel und der zweite Faktor größer als die Wurzel ist. Weiters ist die Multiplikation ja kommutativ, womit die Reihenfolge nicht von Relevanz ist.

Mit dieser Analyse des Problem können wir zu folgendem sprachlich formulierten Algorithmus kommen:

Wenn wir als für alle Zahlen von 2 bis $\sqrt{n}$ testen, ob diese ein Teiler von n ist, dann können wir in *einer* Schleife, in nur *maximal* $\sqrt{n}$ Schleifendurchgängen feststellen, ob eine gegebene Zahl n prim ist oder nicht.

Damit du diesen Algorithmus in `C++` umsetzen kannst, musst du noch wissen wie du zur Wurzelfunktion $\sqrt{}$ kommst. In der Headerdatei `<cmath>` befindet sich der Prototyp dieser Funktion, der für unsere Zwecke folgende Gestalt hat: `double sqrt(unsigned int arg)`. Das kleine ”c” im Namen der Headerdatei `cmath` deutet darauf hin, dass die Headerdatei von der Programmiersprache `C` kommt und für `C++` angepasst wurde, sodass diese direkt zu verwenden ist.

Jetzt bis loslegen und `is_prim` im Modul `primes` implementieren und danach in `main` mit den gleichen Werten wie `is_prime0` testen. Damit die Ausgaben auch eine eine Aussagekraft aufweisen, setze jeweils eine Überschrift vor den jeweiligen Ausgaben, damit der geschätzte Benutzer des Testprogrammes (also du) weiß, welche Ausgaben von welcher Funktion stammen.

8. Gut, der Algorithmus ist jetzt besser als der erste Versuch. Allerdings stellt sich jetzt die Frage wie die Wurzel zu berechnen ist. Okay, wir müssen jetzt nicht die Wurzel einer Zahl selber ziehen. Das macht keinen Sinn, da wir davon ausgehen können, dass die Wurzel effizient in der Standardbibliothek implementiert ist.

   Was dann? Ist das Wurzelziehen einfach? Hmm,… von Hand ist es gar nicht so einfach. Zwar ist der Algorithmus zum Wurzelziehen eindeutig definiert, aber eben mit einem gewissen Aufwand verbunden. Der entscheidende Punkt ist, dass es sich um keine elementare Operation ist, die direkt im Prozessor implementiert ist.

   Also gut wäre es, wenn wir diese Funktion ersetzen könnten. Jetzt ist es wieder Zeit zu überlegen…

   Was bedeutet ”Wurzel einer (ganzen) Zahl”? Das bedeutet, wenn man diese Wurzel mit sich selbst multipliziert wieder die ursprüngliche Zahl erhält: $\sqrt{n} \cdot \sqrt{n} = n$. Uns interessiert allerdings nur der ganzzahlige Anteil der Wurzel (also ohne Nachkommastellen), da wir ja nur mit ganzen Zahlen arbeiten.

   Wenn wir also alle ganzen Zahlen von 2 bis zu der Zahl verwenden, deren Quadrat kleiner gleich als n ist… dann benötigen wir keine Wurzelfunktion mehr, sondern

nur mehr die Multiplikation. Und die Multiplikationsoperation ist im Prozessor direkt als Maschinensprachenbefehl vorhanden ist.

Bitte `is_prime` diesbezüglich anpassen.

9. Nachdem wir das erste Problem jetzt gelöst haben, wenn wir uns dem nächsten Problem zu. Wir wollen alle Primzahlen finden bis zu einer gegebenen oberen Schranke (also einer Zahl wie z.B. 100).

   Schreibe deshalb eine Funktion `void print_primes0(unsigned int n)` im Modul `primes`, die auf Basis unserer Funktion `is_prime` alle Primzahlen kleiner gleich `n` auf der Standardausgabe in einer einzigen Zeile jeweils durch ein Leerzeichen getrennt ausgibt.

   Ready, steady, go!

10. Hmm, auch das ist nicht besonders effizient. Der griechische Mathematiker Eratosthenes (ca. 3.Jh. v. Chr.) hat sich eine Methode einfallen lassen, die als "Sieb des Eratosthenes" bekannt ist. Die Idee ist, alle Zahlen zu sieben, sodass alle Zahlen, die keine Primzahlen sind, durchfallen und nur die Primzahlen bleiben.

    Dieses Sieb funktioniert folgendermaßen:

    a) Schreibe alle natürlichen Zahlen größer gleich 2 in einer Sequenz bis zu einer oberen Schranke auf.

    b) Die erste Zahl dieser Sequenz wird als aktuelle Zahl angesehen.

    c) Nimm die aktuelle Zahl und streich alle Vielfachen dieser Zahl weg.

    d) Nimm die nächste nicht durchgestrichene Zahl als aktuelle Zahl.

    e) Solange die aktuelle Zahl kleiner gleich der Wurzel der oberen Schranke ist, gehe zu Punkt c). Den Grund für die Verwendung der Wurzel kennen wir ja bereits.

    Und nun zur Implementierung. Hierfür wollen wir Arrays verwenden. Den Datentyp `vector` der Standardbibliothek kennen wir schon, aber jetzt geht es darum, die in die Programmiersprache *eingebauten* Arrays einzusetzen.

    Die Größe eines Arrays muss in `C++` zur *Übersetzungszeit* bekannt sein (wenn dieses am Stack angelegt wird) und das funktioniert folgendermaßen, wenn die Größe des Arrays 100 boolesche Werte beinhalten soll:

    ```
    bool is_prim[100];  // Arraygröße muss konstanter Ausdruck sein
    ```

    So, jetzt wollen wir einmal etwas ausprobieren. Implementiere eine Funktion `void print_primes0_upto100()` einmal insoferne, dass diese lediglich eine lokale Variable (wie oben gezeigt) anlegt und alle booleschen Werte je durch ein Leerzeichen getrennt in einer Zeile ausgibt.

    Was siehst du? Ist es wie erwartet?

11. Ja oder nein ist völlig egal. Die Frage ist was man machen kann, dass die Ausgabe so ist wie man sich das erwartet? Eigentlich sollten wir einmal überlegen warum die Ausgabe derart ist, wie diese eben ist…

    Das liegt wiederum daran, dass lokale Variable **nicht** automatisch initialisiert werden. Gehe daher her und ändere die Definition des Arrays wie folgt ab:

    ```
    bool is_prim[100]{};
    ```

    Probiere jetzt dein Programm aus und die Ausgabe wird so sein, wie du diese erwartest.

    Weiter zum nächsten Punkt.

12. Allerdings ist der Wert `false` für alle unsere Werte im Array zwar deterministisch, aber er passt nicht zum Algorithmus und dem Namen der lokalen Variable. Besser wäre es, wenn wir alle Werte in diesem Array mit `true` Werten befüllen will, dann muss man sich selber darum kümmern.

    D.h. alle Arrayplätze durchgehen und mit `true` befüllen. Implementiere jetzt dieses Verhalten.

13. Wir haben jetzt alle Werte mit `true` befüllt. Es macht allerdings keinen Sinn zuerst alle Arrayplätze mit `false` zu initialisieren und danach mit `true` zu überschreiben.

    Die Initialisierung bedeutet, dass der Compiler Code generieren muss, der alle Werte des Arrays mit `false` initialisiert. Es ist zwar so, dass dies sehr effizient erledigt werden kann, aber es ist trotzdem nicht notwendig. Entferne daher wieder die Initialisierung.

14. Wie kann der Algorithmus aber funktionieren? Wir benötigen eigentlich die Zahl und weiters auch noch den Wert ob es sich um eine Primzahl handelt oder nicht.

    Der boolesche Wert ist in unserem Array gespeichert, aber woher nehmen wir die eigentliche Zahl? Natürlich könnten wir ein zweites Array nur mit den Zahlen anlegen, aber das ist überhaupt nicht notwendig, wenn wir den Index im Array gleich der Zahl annehmen.

    Das funktioniert gut, aber wenn wir das so realisieren, dann gibt es einen kleinen Fehler, da die Definition des Arrays jetzt nicht mit der Aufgabenstellung im Einklang ist. Was ist falsch?

    Denken, denken, denken,…

    - Gefunden? Gut und weiter mit dem übernächsten Punkt.
    - Nicht gefunden? Klarerweise nicht gut, dann weiter mit dem nächsten Punkt.

15. In der Aufgabenstellung wird gefordert, dass alle Primzahlen kleiner *gleich* 100 zu ermitteln sind. Wir haben uns in unserer Implementierung aber festgelegt, dass der Index gleich der Zahl sein soll. Bei der Definition eines Arrays in der angegebenen

Art und Weise geht der Index von 0 bis 99. Damit sind wir also nicht mehr in der Lage die Zahl 100 darzustellen. D.h. das Array muss einfach größer sein!

Führe die Änderung durch und weiter mit dem übernächstem Punkt.

16. Gut, dass du herausgefunden hast was der kleine Fehler in unserer Definition ist. Weiter mit dem nächsten Punkt.

17. Nachdem wir den Fehler kennen, können wir auch die Funktion so umändern, dass nicht die booleschen Werte sondern die Zahlen selber ausgegeben werden (in der gleichen Form).

    Gut, jetzt sehen wir also die Zahlen von 1 bis inkl. 100.

18. Klarerweise handelt es sich dabei nicht um die Primzahlen, die du jetzt zu sehen bekommst. Daher machen wir uns gleich daran, den Algorithmus von Eratosthenes zu implementieren.

    Teilen wir den Algorithmus in insgesamt drei Teile:

    a) Initialisieren der Variable `is_prime`

    b) Durchstreichen aller Primzahlen

    c) Ausgabe der Primzahlen

    Den Punkt a) haben wir schon erledigt, den Punkt c) schon angefangen, aber nicht korrekt implementiert. D.h. adaptiere den Code, dass dieser unseren Anforderungen entspricht.

    Klarerweise wird die Ausgabe nach dieser Änderung sich nicht unterscheiden, da der Punkt b) noch nicht implementiert wurde.

19. Bleiben wir noch kurz bei der Ausgabe. Die Ausgabe von 1 ist schon durch die Definition der Primzahlen falsch. Ändere daher die Schleifen entsprechend ab.

20. Jetzt aber zum eigentlichen Durchstreichen der Nicht-Primzahlen. Hier der Algorithmus wieder in `C++`:

```
for (int i{2}; i * i <= 100; ++i) {
    if (is_prime[i]) {
        for (int j{i}; j <= 100 / i; ++j)
            is_prime[i * j] = false;
    }
}
```

    Schaue dir den Algorithmus, stimmt dieser? Wo ist die Wurzel geblieben? Wie sieht das mit der inneren Schleife aus?

21. Ok, setze jetzt diesen Code ein und teste. In Ordnung? Ja, das sollte funktionieren.

22. Jetzt ist es so, dass wir überall 100 bzw. 101 verwendet haben. Wollten wir die Primzahlen bis 200 ermitteln und ausgeben, dann müssten wir überall diese Zahlen ändern. Das ist nicht elegant und auch fehleranfällig.

    Der naheliegende Ansatz ist eine Variable zu verwenden. Ersetze 100 durch `n` und 101 durch `n + 1` und initialisiere eine Variable `n` mit 100. Los geht's.

    Ob dies ohne Syntaxfehler übersetzt, hängt vom Compiler ab. Es kann sein, dass der Compiler es nicht übersetzt und damit den ISO `C++` Standard genau implementiert. Es kann allerdings sein, dass es funktioniert und der Compiler den derzeitigen ISO `C++` Standard in diesem Fall ignoriert.

    Füge in diesem Fall zu deiner `CMakeLists.txt` Datei die folgende Zeile hinzu:

    ```
    add_compile_options(-Wpedantic)
    ```

    Jetzt fehlt nur noch mehr eine Neukonfiguration des CMake-Projektes und dann kannst du wieder übersetzen und eine entsprechende Warnung bewundern.

    Der Grund liegt darin, dass in den Arraygrenzen nur ein konstanter Ausdruck enthalten sein darf. Mehr dazu im nächsten Punkt.

23. Ok, wie auch immer der `C++` Standard im Compiler implementiert ist, was uns interessiert, wie man dem Compiler mitteilen kann, dass es sich bei `n + 1` um einen konstanten Ausdruck handelt.

    Eine Variable kann als "Konstante" deklariert werden, wenn das Schlüsselwort `const` verwendet wird:

    ```
    const int n{100};
    ```

    Das bewirkt, dass die Variable `n` zur Laufzeit mit dem angegebenen Wert 100 initialisiert wird, aber danach nicht mehr verändert werden kann. Damit wird auch die Warnung weggehen!

    Also, nachfolgende Zeile würde der Compiler mit obiger Definition von `n` allerdings bemängeln:

    ```
    n = n + 1
    ```

24. Das funktioniert soweit und deshalb schauen wir uns wieder an, was wir an der Implementierung und am Algorithmus verbessern können.

    Beginnen wir einmal gleich mit der Initialisierung. Wir haben festgestellt, dass der Compiler die Initialisierung in einem Schwung effizienter erledigen kann als wenn wir in einer Schleife alle Arrayelemente einzeln initialisieren. Die Lösung dafür wäre die Bedeutung der Variable `is_prim` zu ändern und auch gleichzeitig den Variablennamen abzuändern.

    Was machen wir eigentlich beim Sieb vom Eratosthenes? Wir streichen alle nicht Primzahlen durch. Wenn wir also die Bedeutung unseres Arrays so abändern, dass wir nicht die Primzahlen repräsentieren sondern die durchgestrichenen Zahlen (die

Nicht-Primzahlen), dann können wir das Array mit lauter `false`-Werten initialisieren und den Job dem Compiler überlassen.

Ändere deshalb den Variablenamen auf `marked` und auch den Algorithmus entsprechend ab (nicht auf die Initialisierung in der Defintion vergessen).

25. Die nächste Verbesserung ist doch ein wenig von grundlegender Art und deshalb werden wir diese in einer neuen Funktion `void print_primes1_upto_100` vornehmen! Daher diese in das Modul `primes` aufnehmen.

    Jeder Index wird als Zahl repräsentiert. Für die Zahlen 0 und 1 gibt es per Definition keine Primzahlen. Damit macht es eigentlich auch keinen Sinn für diese Zahlen Platz im Array zu reservieren. Weiters wissen wir auch von der Zahl 2 per Definition, dass es sich um eine Primzahl handelt. Außerdem ist bekannt, dass alle geraden Zahlen außer der Zahl 2 keine Primzahlen sein können.

    Fassen wir diese Bedingungen alle zusammen, dann reicht es aus, das Sieb anfangs nicht mit allen natürlichen Zahlen zu füllen sondern nur mit den ungeraden Zahlen größer gleich 3!

    Es ist daher ausreichend mit der Sequenz 3,5,7,9,...,99 zu operieren. Damit muss unser Array nicht mehr die Größe 101 sondern nur mehr die Größe $48 = (100 - 3)//2$ aufweisen.

    Allerdings stellt sich jetzt die Frage wie das Durchstreichen funktionieren soll. Probiere einmal auf einem Zettel aus wie das Durchstreichen funktioniert. Also: 3 $\rightarrow$ 9, 15, 21,...; 5 $\rightarrow$ 15, 25, 35,... Was fällt dir auf?

    DENKEN!

    Tipp: Abstand der einzelnen zu durchstreichenden Zahlen?

    DENKEN!

    Und erkannt? Der Abstand ist immer gleich groß wie der Faktor der gerade betrachtet wird.

    Nächste Fragestellung: Wie kann der Faktor aus dem Index berechnet werden?

    DENKEN!

    Und erkannt? Ja? Nein? Egal, bitte sehr: factor $= 2 \cdot i + 3$

    Implementiere jetzt die Funktion `print_primes1_upto100()`. Wenn du es *nicht* zusammenbringst, dann hilft dir der nächste Punkt weiter. Ansonsten gehe zum übernächsten Punkt.

26. Wenn der vorherige Punkt nicht erledigt werden konntest, dann gebe ich dir hier den Programmcode zum Durchstreichen der Zahlen an:

```
// Sieb des Eratosthenes
for (int i{}; i * i <= n; ++i) {
    unsigned int factor = 2 * i + 3;
    if (not marked[i]) {
```

```
            int curr{i};
            while (n - curr > factor) {
                curr += factor;
                marked[curr] = true;
            }
        }
    }
```

27. Falls es noch nicht passiert ist, dann hast du jetzt die Gelegenheit deine Funktion `print_primes1_upto100` in `main` einzubauen. Die Ausgabe sollte genau so sein wie zuvor!

28. Zum Ende zu werden wir noch ein paar strukturelle Änderungen vornehmen und noch einige C++-Features kennenlernen.

    Um die Änderungen im Überblick zu haben, gehe bitte folgendermaßen vor, dass du eine Funktion `print_primes2_upto100` als eine exakte Kopie von `print_primes1_upto100` anlegst und den Aufruf wie schon gehabt in `main` einbaust.

    Als erste Veränderung wollen wir die "Größe" unseres angelegten Arrays ermitteln. Um die Größe eines Speicherobjektes zu ermitteln, kann dies folgendermaßen durchgeführt werden:

    ```
    sizeof(marked)
    ```

    Verändere also die neue Funktion, sodass auch diese Größe ausgegeben wird (wir werden diese Änderung später wieder entfernen).

    Was bekommst du als Ausgabe? Die Anzahl der Elemente des Arrays?

    Verändere die Definition von `marked` so ab, dass es jetzt ein Array von `int` ist. Starte das Programm wiederu. Die Anzahl der Elemente des Arrays?

    Bemerkst du, dass dein Programm genauso funktioniert, obwohl du anstatt eines Array von `bool` Werten ein Array von `int` Werten verwendet hast. Wie schon gelernt, liegt das an der impliziten Konvertierung von `true` zu 1 bzw. von `false` zu 0 und umgekehrt von jedem nicht 0 Wert zu `true` bzw. jedem 0 Wert zu `false`!

    Weiter zum nächsten Punkt.

29. Ok, nicht die Anzahl der Elemente, sondern die Größe des Speicherobjektes in Vielfachem der Größe eines `char` Wertes.

    Füge daher zur *Ausgabe* der Größe folgende Ausgabe *hinzu*:

    ```
    sizeof(marked) / sizeof(int)
    ```

    Jetzt sollte 48 als Ausgabe erscheinen.

    Aha, zuerst die Ausgabe des Speicherobjektes von `marked`, dann die Anzahl der Elemente des Arrays `marked`, da bei der zweiten Ausgabe die Größe des Speicherobjektes `marked` durch die Speichergröße eines Elementes des Arrays (also im Moment vom Typ `int`) dividiert hast.

Ändere jetzt den Typ des Arrays *zurück* auf ein Array von `bool` Werten. Schaue dir die Ausgabe wieder an. Was erkennst du? Wie groß ist also der Speicherplatzbedarf eines `bool` (als Vielfaches eines `char`?)

Wieder etwas gelernt und weiter zum nächsten Punkt.

30. Eine Funktion sollte eigentlich nur eine richtige Aufgabe implementieren. Bei unserer Funktion `print_primes2_upto100` sind es eigentlich 2 Aufgaben:

    - Ermittlung der Primzahlen
    - Ausgabe der Primzahlen

    Wir erstellen deshalb eine Funktion `void sieve(bool[] marked)`, die ein übergebenes Array von `bool` Werten entsprechend dem Algorithmus von Eratosthenes abarbeiten soll.

    Diese Funktion muss nicht unbedingt in der Schnittstelle des Modul `primes` enthalten sein und daher ist es auch nicht notwendig den Prototypen in die Datei `primes.h` aufzunehmen.

    Verschiebe den gesamten Code, der sich mit der Ausgabe der Größe des Arrays als auch den eigentlichen Algorithmus in diese Funktion und rufe diese Funktion mit dem initialisierten Array `marked` auf. Eine Kleinigkeit ist zu beachten: Die Definition der Variable, die die Anzahl der Elemente beinhaltet muss ebenfalls in die Funktion *kopiert* werden. Das ist natürlich nicht schön und wir werden uns gleich darum kümmern.

    Ok, damit du einen Überblick bekommst wie so etwas aussehen kann, hier der Sourcecode der Funktion:

```
void sieve(bool marked[]) {
    const int n1{48};

    cout << sizeof(marked) << endl;
    cout << sizeof(marked) / sizeof(bool) << endl;

    for (int i{}; i * i <= n1; ++i) {
        unsigned int factor = 2 * i + 3;
        if (not marked[i]) {
            int curr{i};
            while (n1 - curr > factor) {
                curr += factor;
                marked[curr] = true;
            }
        }
    }
}
```

Wenn du das jetzt übersetzt wirst du Warnungen von dem Compiler bekommen! Schaue dir die Warnungen an (d.h. nicht nur schauen auch *lesen*) und starte das Programm. Dann versuche die Warnungen des Compilers mit der Ausgabe in Einklang zu bringen.

Die Aufklärung dieses Falles erfolgt im nächsten Punkt.

31. Die Sache ist die… Ein Array wird in `C++` als Pointer auf das erste Element übergeben!!! Da ändert auch die wohlgemeinte Schreibweise als Array irgendetwas. Da es sich um einen Pointer handelt wirst du auf einem 32 Bit-System jeweils die Ausgabe "4" erhalten haben.

    Das wiederum bedeutet, dass wir in der Funktion *keinen* Zugriff auf die Größe des Arrays haben.

    Derzeit haben wir die Variable mit der Größeninformation in die Funktion kopiert, damit wir die Größeninformation innerhalb der Funktion zur Verfügung haben und das Beispiel funktioniert.

    Das ist natürlich keine Lösung. Die einzige Lösung ist, nicht nur das Array zu übergeben sondern auch die Größe.

    Ändere sowohl die Signatur der Funktion ab als auch den Aufruf und auch die Implementierung, damit eben die Größe übergeben wird.

    Bei dieser Gelegenheit kannst du gleich die Ausgabeanweisungen in der Funktion entfernen, da die Compilerwarnungen nun doch nicht so anregend sind und eigentlich waren die Ausgabeanweisungen nur dafür da, dir die Größe und Übergabe von Arrays näherzubringen.

32. Kurz noch eine Denkpause, also keine Pause vom Denken, sondern eine Pause vom Codieren zum Denken `;-)`: Warum haben wir das Array nicht mehr als Rückgabewert zurückgeben müssen und es trotzdem in der aufrufenden Funktion korrekt verwenden können?

    Ja, genau, so ein Pointer ist in diesem Zusammenhang wie eine Referenz in Python.

33. So weiter mit dem Codieren. Eigentlich sind noch zwei unschöne Kleinigkeiten in unserem Code enthalten:

    - Eigentlich muss es sich bei Größenangaben eines Arrays gemäß aktuellem `C++` Standard in Arraydefinitionen um konstante Werte handeln. Das ist ja auch der Grund warum wir (bei Angabe der Compileroption `-Wpedantic`) derzeit eine Warnung bekommen.
    - Außerdem ist es nicht wirklich hübsch eine Funktion zu haben, die auf die Ausgabe der Primzahlen kleiner gleich 100 beschränkt ist. Eigentlich will man eine Funktion haben, der man die Obergrenze als Argument beim Aufruf mitgeben kann.

Wir werden uns dieser beiden Punkte jetzt zum Schluss annehmen.

Schreibe eine Funktion `void print_primes(unsigned int n)`, die wie die Funktion `print_primes2_upto_100` funktioniert, abgesehen von der variablen Obergrenze.

34. Die Warnung haben wir allerdings noch immer. Gehe daher her und ersetze die Definition des Arrays mit folgender Pointerdefinition (samt Initialisierung):

    ```
    bool* marked{new bool[n1]{}};
    ```

    Übersetze und starte dein Programm. Was fällt dir auf?

    Ist dir aufgefallen, dass

    - kein Fehler vom Compiler produziert wurde?
    - keine Warnung mehr vom Compiler ausgegeben wurde?
    - das Programm wie vorher funktioniert?

    Gut, aber wieso ist das so?

    - Es wurde kein Fehler produziert, weil ein Array als Pointer auf das erste Element übergeben wird und wir einen Pointer übergeben haben.
    - Es wurde keine Warnung produziert, weil eben keine Konstante in einer Speicherplatzanforderung mittels eines `new` Ausdruckes notwendig ist (gemäß des aktuellen `C++` Standards).
    - Funktioniert hat es, weil es für die Funktion `sieve` völlig egal ist, ob das Array am Stack oder am Heap liegt.

35. Was dir nicht aufgefallen ist, dass es eigentlich noch immer einen Fehler beinhaltet, aber das schauen wir uns jetzt an.

    Der Fehler ist, dass wir ein Array am Heap angelegt haben, dieses aber nicht gelöscht haben. Wie gelernt werden alle Speicherobjekte, die am Stack liegen von `C++` automatisch wieder gelöscht. Dies betrifft aber nicht die Speicherobjekte am Heap. Diese müssen selber verwaltet werden.

    Wenn dies aber ein Fehler ist, dann stellt sich die Frage warum dieser nicht in Erscheinung getreten ist. Die Antwort ist auf zweierlei zu beantworten:

    a) Nicht jeder Fehler muss immer in Erscheinung treten! Irgendwann tritt jeder Fehler irgendwann in Erscheinung, aber eben nicht bei jedem Programmlauf.

    b) Der eigentliche Grund ist in diesem konkreten Fall aber ein anderer: Das laufende Programm (der Prozess) beendet sich unmittelbar nach dem Aufruf von `print_primes`. Bei Beendigung eines Prozesses wird automatisch der gesamte Speicher des Prozesses vom Betriebssystem freigegeben.

Vielleicht könntest du einwerfen, dass der Pointer `marked` am Stack liegt und deshalb von `C++` wieder gelöscht wird. Das ist richtig, denn der Speicher von `marked` wird auch freigegeben, aber nicht der Speicher auf den `marked` zeigt!

Problem erkannt, Gefahr gebannt… Lösche den Speicher manuell am Ende der Funktion `print_primes` mittels:

```
delete[] marked;
```

Die Erklärung warum `delete[]` und nicht `delete` zu verwenden ist? Wenn ein Array mittels `new` angelegt worden ist, dann **muss** das Array auch mit `delete[]` wieder freigegeben werden.

Vielleicht fragst du dich, warum `C++` das nicht selber weiß? Ja, dann folgt hier zum Schluss noch die Antwort: Ein Array wird als Pointer auf das erste Element angesehen, richtig? Und woher soll `C++` wissen, ob nur der Speicherbereich für das erste Element oder der gesamte Speicherbereich des Array freigegeben werden soll.

Fertig ist das Programm.

4 Übungszweck dieses Beispiels

- Primzahlen
- `bool`, `ios::boolalpha`, `ios::noboolalpha`
- CMake (mit src, include und build)
- Algorithmus analysieren und schrittweise verbessern
- Sieb des Eratosthenes
- einfache Arrays
- `const`
- Größe vs. Speichergröße eines Array, `sizeof`
- Übergabe eines Array an eine Funktion
- Arrays am Heap: `new` und `delete[]` vs. `delete`