Fachhochschule Nordwestschweiz

Programmieren in C++

Dr. Daniel Wolleb / Prof. Dr. C. Stamm

Übung 1: Audiodatenverarbeitung

Lernziele

- Sie repetieren Stoff aus der Vorlesung: Struktur eines C++-Programms, einfache Datentypen, Arrays, Referenzen, Zeigerarithmetik, Unveränderbarkeit.
- Sie lesen und schreiben Dateien im Binärformat.
- Sie berechnen einfache Audioeffekte.
- Sie beschleunigen ein bestehendes Programm mittels einer Präfixsummendatenstruktur.
- Sie führen Korrektheits- und Laufzeitmessungen durch.

Überblick

In dieser Übung schreiben Sie ein C++-Programm, welches Audiodateien im WAVE-Format einlesen, bearbeiten und wieder schreiben kann. Das WAVE-Format unterstützt unkomprimierte Audioaufnahmen, das heisst der Datenstrom beschreibt direkt das abgetastete Audiosignal in einer hohen, fixen zeitlichen Auflösung. Es wird also direkt die Wellenform beschrieben, mit welcher die Lautsprechermembran nach innen und aussen schwingen soll. Im Falle einer Stereoaufnahme wird dabei einfach abwechselnd die Position des linken und des rechten Lautsprechers beschrieben.
Zuerst kümmern Sie sich um das korrekte Einlesen des Dateikopfes und der Audiodaten. Als eine erste Überprüfung dessen programmieren Sie eine Visualisierung der Audiowellenstärken auf der Konsole. Als zweite Überprüfung implementieren Sie das Schreiben von Audiodaten im selben WAVE-Format. Wir wollen aber nicht einfach die gleiche Tondatei einlesen und wieder schreiben, sondern ein paar Effekte hinzufügen. Mit nicht allzu viel Aufwand generieren Sie einen Echo-Effekt und können das Resultat Ihres Programmes akustisch überprüfen. Zum Schluss wollen wir noch Pausen in der Tondatei überspringen. Dazu ist eine relativ langsame Implementierung bereits gegeben und Ihre Aufgabe wird es sein, sie deutlich schneller zu machen.

WAVE-Datei einlesen

In der Vorlagendatei *audio.cpp* finden Sie für jede Teilaufgabe einen *TODO*-Kommentar, der die Stelle markiert, an welcher Sie das Programm ergänzen sollen. Dabei lassen sich die Teilaufgaben eine nach der anderen lösen, das heisst *audio.cpp* kompiliert bereits von Beginn weg und Sie können Ihren Fortschritt immer wieder überprüfen.
Schauen wir uns zuerst die *main*-Funktion an:

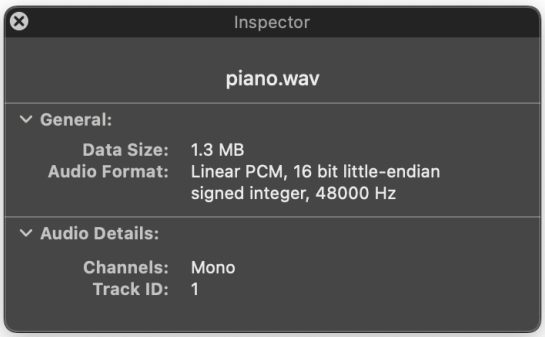
```
int main(int argc, const char * argv[]) {
    string command = argv[1];
    string inputFilename = argv[2];
    Header header;
    int32_t sampleRate;
    int32_t sampleCount;
    int16_t channelCount;
    vector<vector<int16_t>> samples = read(inputFilename, header, sampleRate, sampleCount, channelCount);
```

```
if (command == "info") {
  //...
```

Der Aufruf des Programms erwartet also zwei Argumente: einen Befehl und einen Dateinamen. Unabhängig vom Befehl wird anschliessend die *read*-Funktion aufgerufen, welche, den Argumenten nach anzunehmen, die Datei mit dem gegebenen Namen einliest, und zwar in den *samples*-Vektor unter Angabe des Dateikopfes, der Anzahl Samples, der Anzahl Kanäle und der Abtastrate. *read* hat demnach nur ein Eingabeargument, vier Rückgabewerte in Form von Referenzvariablen und den *samples*-Vektor als Rückgabewert.

Anschliessend folgen die verschiedenen Funktionsaufrufe abhängig vom Befehl. Sie sind entsprechend der Teilaufgaben geordnet. Wir beginnen mit dem Befehl *info*, der lediglich einige Eckdaten zur Audiodatei sowie einige der Samples ausgibt.

Im Vorlageordner zu dieser Übung finden Sie eine kurze Audiodatei namens *piano.wav*. Hören Sie sich die Aufnahme kurz an, sie dauert nur 13.5 Sekunden. Ihr Audioplayer wird Ihnen vielleicht bereits einige Informationen zur Datei anzeigen:



So ist die Abtastrate 48 Kilohertz, jedes Sample belegt 16 Bit an Speicherplatz und die Datei verfügt nur über eine Tonspur (mono). Bei 13.5 Sekunden ergibt das also 13.5 · 48’000 · 16 / 8 = 1.296 Megabytes. Mit *Linear PCM* wird schliesslich noch beschrieben, wie das Audiosignal digitalisiert wurde. *PCM* steht für *Pulse-Code-Modulation*, was das Vorgehen bezeichnet, ein Audiosignal in regelmässigen Abständen, in unserem Fall 48’000 Mal pro Sekunde, abzutasten und jeweils die aktuelle Amplitude, also die relative Auslenkung der Lautsprechermembran, in einer fixen Anzahl Bits, in unserem Fall 16 Bit, zu quantisieren. Das *Linear* schliesslich bezeichnet, dass bei dieser Quantisierung eine lineare Skala angewendet wurde. Das heisst, wenn die Membran des Lautsprechers maximal einen Zentimeter von der Nullposition ausgelenkt werden kann, dann wird dieser Zentimeter in 2^{15} = 32’768 gleiche Teile aufgeteilt und das Signal entsprechend kodiert. Da die Membran in beide Richtungen ausschwingt, werden auch die Werte bis -2^{15} benötigt und entsprechend ein *signed integer* verwendet.

Wissend, was uns in der Datei erwartet, können Sie sich an die Implementierung der *read*-Funktion machen. Die Vorlage enthält bereits den Funktionsaufruf, um die Datei zu öffnen:

```
ifs.open(filename, ios::binary);
```

Für das eigentliche Einlesen der Datei benutzen wir nun die Funktion *ifs.read*. Diese Funktion akzeptiert zwei Argumente: eine Speicheradresse, wo die eingelesenen Werte hingeschrieben werden sollen und die Anzahl zu lesender Bytes.

Wie in der Vorlage gezeigt, lassen sich so auch ganze Strukturen direkt einlesen mittels:

```
struct Header {
  char riff[4];
  uint32_t fileSize;
  // ...
};
// ...
ifs.read(reinterpret_cast<char*>(&header), sizeof(Header));
```

Zur Fehlersuche kann es sehr hilfreich sein, die eingelesenen Werte gleich wieder auf die Konsole auszugeben oder mittels *assert* auf den erwarteten Wert zu überprüfen. Jede WAVE-Datei beginnt mit den vier Buchstaben *RIFF*, was für *Resource Interchange File Format* steht, weshalb wir das in der Vorlage auch gleich ausgeben und überprüfen:

```
cout << "RIFF: " << header.riff[0] << header.riff[1] << header.riff[2] << header.riff[3] << endl;
assert(header.riff[0] == 'R' && header.riff[1] == 'I' && header.riff[2] == 'F' && header.riff[3] == 'F');
cout << "file size: " << header.fileSize << endl;
```

Beachten Sie, dass in unserem *riff*-Array kein Platz für das übliche Null-Byte am Ende eines C-Strings bleibt und wir ihn daher nur Buchstaben für Buchstaben und nicht als Ganzes ausgeben können.
Am Ende der Funktion werden die Samples von einem rohen Pointer in einen zweidimensionalen Vektor namens *rearrangedSamples* umarrangiert, also in einen Vektor von Vektoren. Schauen wir uns diesen Teil kurz an:

```
vector<vector<int16_t>> rearrangedSamples(channelCount);
uint32_t s = 0;
for (int32_t sample = 0; sample < sampleCount; sample++) {
  for (int16_t channel = 0; channel < channelCount; channel++) {
    rearrangedSamples[channel].push_back(samples[s++]);
  }
}
```

Die Audiodaten werden also so umgruppiert, dass mit *rearrangedSamples[i][j]* anschliessend auf das j-te Sample im i-ten Kanal zugegriffen werden kann.

Aufgabe 1

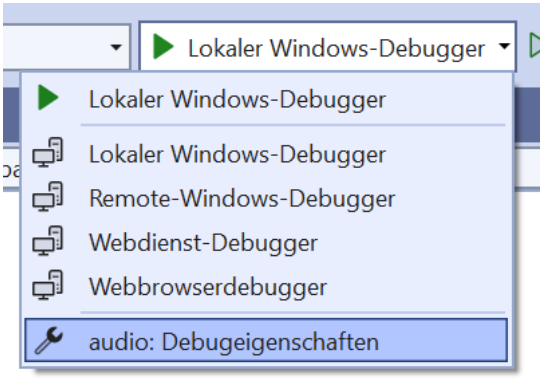
Vervollständigen Sie die Header-Struktur und die *read*-Funktion. Folgende Tabelle zeigt die Struktur des Dateikopfes auf:

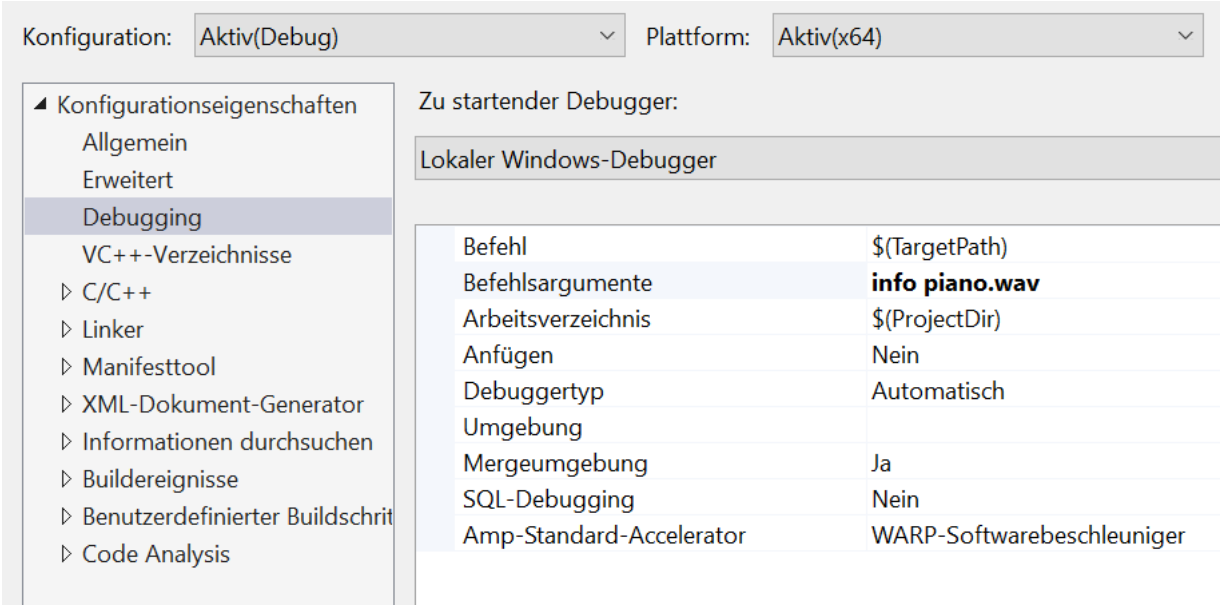
Grösse	Inhalt
4 Bytes	Zeichenkette “RIFF”.
4 Bytes	Dateigrösse in Bytes minus 8.
4 Bytes	Zeichenkette “WAVE”.
4 Bytes	Zeichenkette “fmt “ (inklusive Leerzeichen am Ende).
4 Bytes	Länge des restlichen Format-Headers in Byte. Sollte 16 sein.
2 Bytes	Formattyp. 0x0001 steht für lineares PCM, etwas anderes unterstützen wir nicht.
2 Bytes	Anzahl Kanäle.
4 Bytes	Abtastrate in Hz, also Anzahl Samples pro Sekunde. Z.B. 44’100 oder 48’000.
4 Bytes	Anzahl Bytes pro Sekunde.
2 Bytes	Anzahl Bytes pro Sample für alle Kanäle zusammen.
2 Bytes	Anzahl Bits pro Sample pro Kanal. Hier unterstützen wir nur den Wert 16.
4 Bytes	Zeichenkette “data”.
4 Bytes	Anzahl Bytes im folgenden Datenblock.
viele Bytes	Datenblock (nicht mehr Teil des Dateikopfes) mit genau so vielen Bytes wie im vorherigen Feld angekündigt. Dabei stellen immer zwei Bytes zusammengenommen ein Sample dar und sind enkodiert als ein 16-Bit-Integer mit Vorzeichen. Bei zwei Kanälen folgt zunächst das erste Sample für jeden Kanal, dann das zweite Sample für jeden Kanal, usw.

Beachten Sie, dass die read-Funktion zusätzlich zu der Header-Struktur noch drei weitere Referenzvariablen als Funktionsparameter hat. Sie können diese nach dem Einlesen des Dateikopfes aus der Header-Struktur kopieren bzw. ausrechnen.
Für den Datenblock nach dem Header müssen Sie ein genügend grosses Array allozieren. Verwenden Sie dazu einen unique_ptr, welcher auf ein Array des Typs int16_t auf dem Heap zeigt.
Wenn Sie mit dem read-Code zufrieden sind, können Sie ihn wie folgt kompilieren und ausführen:

```
> g++ audio.cpp -o audio -std=c++20
> ./audio info piano.wav
```

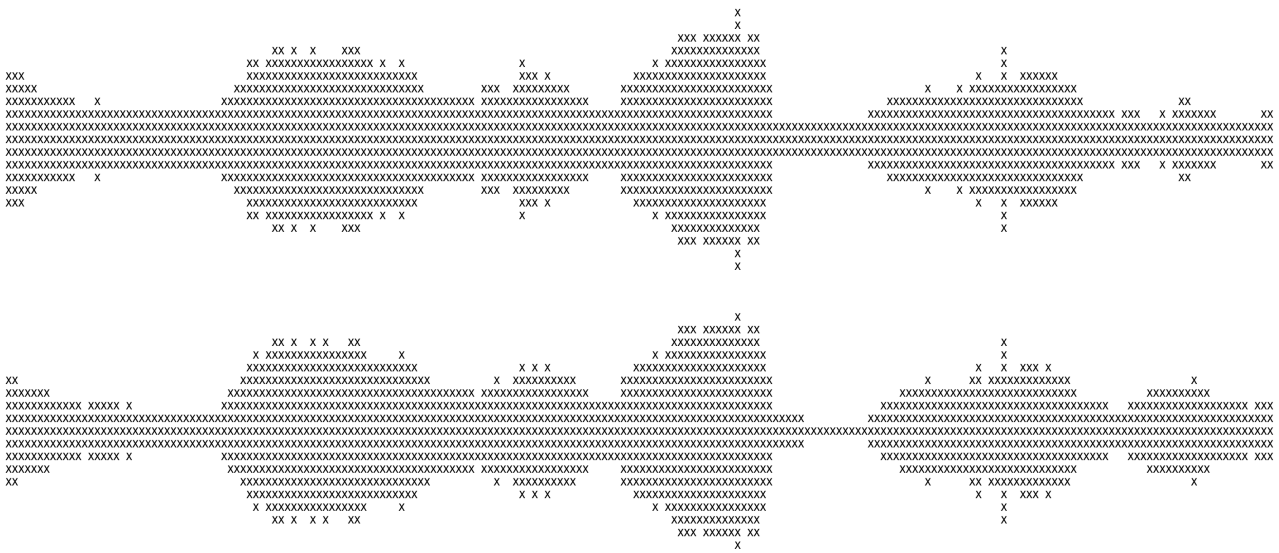
Falls Sie das Programm nicht auf der Konsole sondern in Visual Studio ausführen können Sie die Argumente info piano.wav wie folgt an das Programm übergeben: Klicken Sie beim grünen Startknopf auf den kleinen Pfeil nach unten und dann auf “Debugeigenschaften”. Unter “Konfigurationseigenschaften” → “Debugging” → “Befehlsargumente” können Sie anschliessend die gewünschten Argumente ergänzen und mit dem Knopf “Übernehmen” bestätigen.





Audiowellenformen visualisieren

Als nächstes wollen wir die Audiodaten, die wir soeben eingelesen haben, graphisch darstellen. Dazu ermitteln wir in kurzen Zeitabschnitten jeweils die durchschnittliche Geschwindigkeit, mit der sich die Lautsprechermembran bewegt, um zu bestimmen, wie laut sich der entsprechende Abschnitt anhört. Wenn wir dies für viele aufeinanderfolgende Zeitabschnitte tun, erhalten wir einen Verlauf der Lautstärke, der sich hübsch zweidimensional plotten lässt. Das sieht bei einer Stereoaufnahme dann etwa so aus:



Die Funktion *play* ist bereits vollständig implementiert und enthält alles, um aus einer solchen "Lautstärkenzusammenfassung" pro Kanal obigen ASCII-Plot herzustellen und ihn fünf Mal pro Sekunde neu zu zeichnen. Das einzige was noch fehlt ist die Funktion *summarize*, welche aus den rohen Audiodaten diese Zusammenfassung erstellt.

```
vector<int> summarize(const vector<int16_t> &samples, int from, int until,
                      int numBuckets) {
  vector<int> result(numBuckets);
  // Todo.
  return result;
}
```

Aufgabe 2

Implementieren Sie die Funktion summarize. Die Funktion erhält alle Samples eines Kanals als Vektor sowie zwei Indices from und until, welche den Bereich beschreiben, den es zusammenzufassen gilt. Das letzte Argument numBuckets, gibt die Anzahl Zeitabschnitte vor, die unsere Zusammenfassung enthalten soll.
Unterteilen Sie also den Bereich [from, until) in numBuckets viele, möglichst gleich grosse Abschnitte und berechnen Sie in jedem dieser Abschnitte die durchschnittliche Geschwindigkeit der Lautsprechermembran, also die durchschnittliche, absolute Differenz zwischen zwei aufeinanderfolgenden Samples.
Beachten Sie, dass Ihr Code auch mit ungültigen Indices umgehen können muss, da from zu Abspielbeginn vor das erste Sample zeigt und until am Ende hinter das letzte Sample verweist. Dabei soll der Durchschnitt nur über die existierenden Samples bestimmt werden, das heisst in gewissen Buckets wird der Durchschnitt aus deutlich weniger vielen Werten bestimmt und in einigen Buckets aus gar keinen (in welchem Fall der Durchschnitt gleich Null sein soll).
Testen Sie Ihren Code mittels:

```
> g++ audio.cpp -o audio -std=c++20
> ./audio play Ring10.wav
```

Passt die Ausgabe, die 200 Zeichen breit ist, nicht gut auf Ihre Konsole, dann ändern Sie die Darstellungsgrösse Ihres Terminals oder passen Sie die Anzahl Spalten oder Zeilen in der Vorlage der play-Funktion entsprechend an. Erstellen Sie einen Screenshot an einer guten Abspielposition für die Abgabe.

Echo-Effekt

Nun wollen wir einen ersten Audioeffekt zur eingelesenen Datei hinzufügen, nämlich ein Echo. Bevor wir zum amüsanten Teil, dem Herumexperimentieren mit verschiedenen Verzögerungen und Lautstärken, kommen, ist noch ein kleines Pflichtstück nötig: das Schreiben der Samples in eine neue WAVE-Datei.

Aufgabe 3

Implementieren Sie die Funktion write. Sie können sich dabei stark an Ihrer Implementierung von read orientieren. Die Argumente von ofs.write sind analog zu ifs.read.
Die write-Funktion erhält als letztes Argument auch eine Kopie des Headers der ursprünglich eingelesenen Datei. Passen Sie darin die Dateigrösse und die Datenblockgrösse entsprechend dem samples-Vektor an, um einen für die neue Datei gültigen Header zu erhalten. Sie können diesen neuen Header mit einem einzigen write-Aufruf in die Datei schreiben.
Achten Sie beim Abspeichern der Samples darauf, dass die Reihenfolge analog zum Lesen der Samples angepasst werden muss. Das notwendige Umordnen können Sie direkt beim Schreiben der einzelnen Samples vornehmen.

Um zu testen, ob die Ausgabe korrekt funktioniert, modifizieren Sie die Funktion addEcho so, dass als outputSamples einfach eine 1:1-Kopie der inputSamples zurückgegeben wird. Rufen Sie die Funktion mit dem folgendem Befehl auf – die neu geschriebene Datei echo_ring10.wav sollte sich genau gleich anhören wie das Original:

```
> g++ audio.cpp -o audio -std=c++20
> ./audio echo ring10.wav
```

Aufgabe 4

Fügen Sie nun einen Echo-Effekt zur Funktion addEcho hinzu. Dazu soll das i-te Sample in der Ausgabe aus einer gewichteten Kombination aus dem i-ten Sample und dem (i minus x)-ten Sample der Eingabe bestehen. Dabei können wir zum Beispiel x gleich der halben Abtastrate wählen, um ein Echo mit einer halben Sekunde Verzögerung zu generieren. Als Gewichte für diese Kombination eignen sich beispielsweise ein Gewicht von 2/3 für das i-te Sample und ein Gewicht von 1/3 für das (i minus x)-te Sample. So bleibt die Gesamtlautstärke dieselbe und das Echo ist deutlich leiser als das Originalsignal.
Achten Sie in Ihrem Code darauf, nicht auf ungültige Array-Indices zuzugreifen, das heisst in den ersten x Samples lässt sich noch kein Echo-Sample hinzuaddieren. Bearbeiten Sie alle Kanäle unabhängig voneinander auf die gleiche Art und Weise. Probieren Sie einige Verzögerungen und Gewichtungen aus und hören Sie sich das Resultat an. Halten Sie echo_piano.wav für die Abgabe bereit.
[Wer möchte, kann hier gerne noch weitertüfteln und den Effekt mit mehreren Echos oder mit unterschiedlichen Echos auf verschiedenen Kanälen erweitern. Auch mit der Geschwindigkeit lässt sich gut spielen. So wird die Aufnahme doppelt so schnell (und hoch), wenn wir nur noch jedes zweite Sample verwenden.]

Pausen überspringen

[Diese letzte Aufgabe ist etwas kniffliger und daher optional. Wir ermutigen Sie, dass Sie versuchen diese Aufgabe zu lösen. Für das Erfüllen der Testatpflicht ist das Lösen dieser Aufgabe jedoch keine Voraussetzung.]
Viele Audiobearbeitungsprogramme bieten die Funktion an, Pausen in der Audiodatei automatisch herauszuschneiden. Bei Musikstücken, die dadurch aus dem Rhythmus geraten können, macht das evtl. wenig Sinn. Bei Sprachaufnahmen, wie Interviews, Podcasts oder Hörbüchern, kann dies jedoch sehr nützlich sein, um Sprechpausen zu unterdrücken.
Die Funktion *suppressPause* implementiert solch einen Pausenunterdrückungsfilter. Lesen Sie sich den Code dieser Funktion genau durch. Ähnlich wie in *summarize* aus Aufgabe 2 berechnet *suppressPause* die Lautstärke als Durchschnittsgeschwindigkeit des Audiosignals. Dazu wird für jedes Sample die Zehntelssekunde darum herum (50 Millisekunden davor und 50 Millisekunden danach) in Betracht gezogen und diese Durchschnittsgeschwindigkeit bestimmt. Anschliessend wird der Durchschnitt dieser Durchschnitte bestimmt. Diese Gesamtlautstärke der Audiodatei dient dann als Entscheidungsgrundlage, um zu bestimmen, was leise ist. In der Ausgabe werden alle Samples übersprungen, deren Lautstärke unterhalb von 20% dieser Gesamtlautstärke liegen.
Der Befehl *shorten* ruft die Funktion *suppressPause* indirekt durch die Funktion *timed* auf. *timed* gibt die Laufzeit der Effektfunktion auf die Konsole aus. Wenn Sie also

```
> g++ audio.cpp -o audio -std=c++20
> ./audio shorten Ring10.wav
```

ausführen, dann sollten Sie nach etwas Wartezeit sehen, wie lange der Aufruf von *suppressPause* dauert. Aber obwohl *piano.wav* mit 14 Sekunden Länge relativ kurz ist, dauert die Berechnung dieses Effekts einige Sekunden. Dies liegt einerseits daran, dass wir dem Kompiler keinen Optimierungsauftrag erteilt haben und andererseits an der grossen Anzahl an Additionen, welche in den beiden verschachtelten Schleifen ausgeführt werden müssen. Lassen Sie uns Ersteres beheben: Mit

dem Flag “-O3” werden alle Optimierungen, die GCC und Clang unterstützen, aktiviert. Führen Sie also

```
> g++ audio.cpp -o audio -std=c++20 -O3
> ./audio shorten Ring10.wav
```

aus und seien Sie nicht überrascht, wenn so die Laufzeit um einen Faktor zehn oder mehr verkürzt wird. In Visual Studio wechseln Sie dazu einfach auf die Release-Konfiguration und setzen die Befehlsargument erneut in den Debugeigenschaften.

Wenn wir nun bedenken, dass *piano.wav* rund 650’000 Samples enthält und dass wir im Zehntelssekundenzeitfenster rund 48’000/10 = 4’800 Samples aufsummieren, ergibt das rund 650’000 · 4’800 = 3’120’000’000 Additionen für die *sum*-Variable. Für die *count*-Variable ergibt das ebenso viele Inkrementierungen, die in den paar Sekunden, die der Aufruf selbst mit aktivierten Optimierungen immer noch dauert, durchgeführt werden müssen. Solange wir also mit Prozessoren vorlieb nehmen müssen, deren Taktraten im einstelligen Gigahertzbereich liegen, können wir nicht erwarten, dass dieser Code in Sekundenbruchteilen ausgeführt werden kann.

Um also auch längere Audiodateien, wie zum Beispiel die Datei *mani.wav*, die ebenfalls in der Vorlage enthalten ist, rasch bearbeiten zu können, braucht es eine algorithmische Verbesserung statt bloss eines clevereren Compilers oder einer teureren CPU. Bei genauerem Hinschauen fällt uns auf, dass sich die Summen, welche in *suppressPause* berechnet werden, einander sehr ähnlich sehen. Es werden immer Summen über einen zusammenhängenden Bereich der Audiodatei gebildet – das ideale Einsatzgebiet für die Datenstruktur der Präfixsummen!

In der Präfixsummendatenstruktur wandeln wir eine Eingabezahlenfolge a_0, a_1, a_2, …, a_n um in die Folge $s_{-1} = 0$, $s_0 = a_0$, $s_1 = a_0+a_1$, $s_2 = a_0+a_1+a_2$, …, $s_n = a_0+a_1+a_2+…+a_n$, sodass wir die Summe für einen beliebigen Bereich $[a_i, …, a_j]$ als eine einzige Differenz, nämlich $s_j - s_{i-1}$, ausdrücken können. Mit Zugriff auf die Präfixsummen lässt sich jede Bereichssumme also in konstanter (statt linearer!) Zeit berechnen. Der Aufbau dieser Präfixsummen dauert dabei einmalig lineare Zeit, wenn wir ausnützen, dass $s_i = s_{i-1} + a_i$ gilt.

Aufgabe 5 (optional)

Implementieren Sie eine effizientere Version von suppressPause, indem Sie in der Funktion suppressPausePrefixSum eine Präfixsummendatenstruktur aufbauen und diese anschliessend benutzen, um die innere Schleife der Berechnung der averages aus suppressPause zu eliminieren.

Überlegen Sie sich dabei, welchen Datentyp Sie für die Präfixsummen verwenden müssen, um Überläufe zu verhindern.

Übernehmen Sie den Rest der Logik von suppressPause und stellen Sie mithilfe der Ausgabe von timed sicher, dass Ihre Implementierung deutlich schneller ist und dennoch in der selben Anzahl Samples resultiert. Hören Sie sich die Ausgaben in short_fast_piano.wav und short_fast_mani.wav an, um sicherzugehen, dass die Pausen in der Audiodatei wirklich unterdrückt werden und dass sie den Dateien short_slow_piano.wav und short_slow_mani.wav entsprechen.

Erstellen Sie einen Screenshot der Ausgabe von

```
> g++ audio.cpp -o audio -std=c++20 -O3
> ./audio shorten Ring10.wav
```

Abgabe

Senden Sie die beiden Screenshots aus Aufgabe 2 und Aufgabe 5, die Datei *echo_ring10.wav*, sowie Ihren Quellcode als ungezippte E-Mail-Anhänge an christoph.stamm@fhnw.ch.