

Laboratory 6

Structures and Defining New Types

6.1 Overview

In laboratory 5.1 you saw how arrays could be used to collect information together in a useful way. This lab introduces *structures*, which are another way of collecting information together. In an array, all the entries must be of the same type. The information stored in a structure may be of many different types. Structures allow you to write more compact programs and make them much easier to design and understand.

This laboratory will also introduce you to the `typedef` statement, which allows you to define your own types (like `int`, `double` etc.). This is most useful when used in combination with structures.

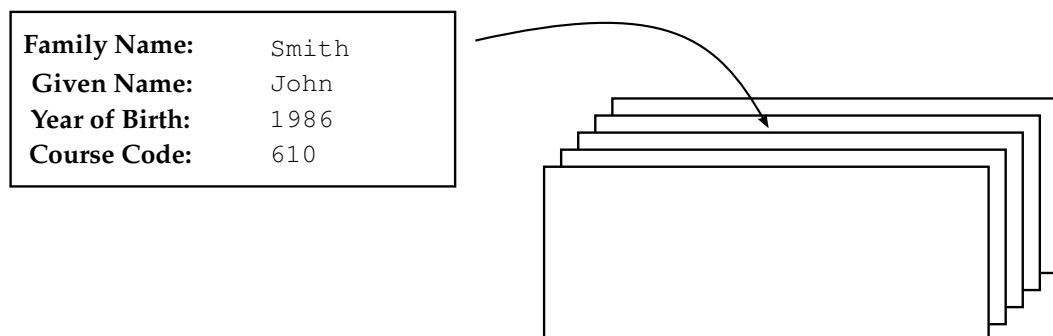
The graphics and music options each show you one way in which structures can be put to good use.

6.2 Introducing Structures

Imagine that we want to write a program to store details about students attending a course at a university. Let us say that we want to be able to store the following pieces of information about the student:

- their family name (or surname);
- their given name (or first name);
- the year in which they were born;
- the code of the course they are on.

Before computers were used to store this kind of information it would be common to put each of these details on a card. The card could then be filed with other cards which have the details of other students.



A card is useful because it keeps the various pieces of information about the student all in one place.

In C we have used arrays to keep pieces of information together, like distances or positions or notes in a musical scale. In an array, all the items must have the same type. This can be very limiting. The details we want to store about each student are not naturally all the same type:

- the family and given names are character strings;
- the year in which they were born is an integer;
- the course code is a three digit integer.

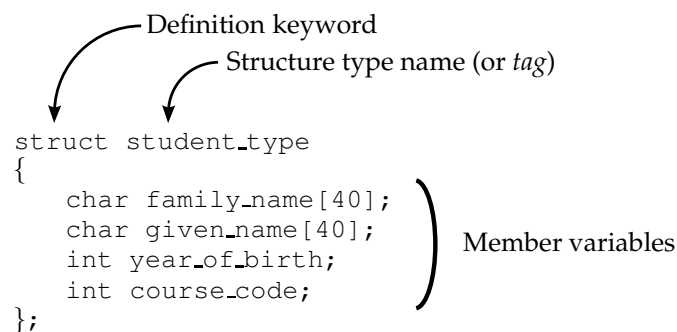
There are two strings and two integers. It would be possible to store the two integers as strings, but this would require conversions to store and retrieve the information. This would be inefficient and error prone.

C *structures* allow us to collect information together in a very similar way to the cards mentioned above. We can declare a structure using the `struct` keyword. This is normally done in two stages:

- Firstly you use the `struct` keyword to define a template for the structure. This is a bit like creating a card which has headings, but has not been filled in yet. The template is given a name so that you can refer to it.
- Once you have declared the template, you can now use it. To do this, you specify the name of the template when declaring a variable in place of a type name like `int` or `double`.

We will look at each of these stages in turn.

To define the template you first specify the `struct` keyword followed by a *tag* which is the name of the template you are defining. Following this, in a set of curly braces, you define one or more variables which are part of the structure. These are called *member variables*. A structure template definition for the details we want to store about a student is shown below.



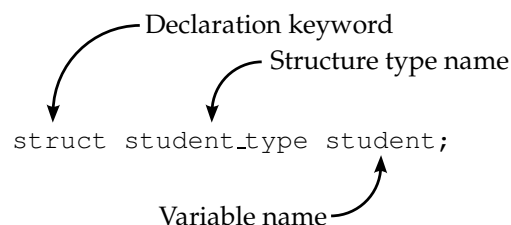
```

struct student_type
{
    char family_name[40];
    char given_name[40];
    int year_of_birth;
    int course_code;
};

```

Remember that this **does not actually declare a variable yet**. All we have done is decided on a template, a layout for the card which we are going to fill in.

Now that we have a template, we can use it. We declare a variable called `student` which is a structure of the template `student_type`:



```

struct student_type student;

```

To use the `student` variable you must specify which one of the member variable you wish to access. For example, to set the year of birth to 1986, you would write:

```
student.year_of_birth = 1986;
```

The dot says that the variable before it is a structure, and the variable afterwards specifies which part of the structure we want to access.

Syntax: Structures

```
struct name
{
    member variable declarations
};
```

A structure is a way of grouping variables together where the variables may have different types. Structures also create more readable code because, unlike arrays, each item of data in the structure has a name, rather than just a number.

A structure definition specifies a new structure type called *name* which has member variables specified by the *member variable declarations*. For example, the following code defines a new structure type called `Point` with two member variables, `x` and `y`.

```
struct Point
{
    int x;
    int y;
};
```

This structure definition can also be written:

```
struct Point
{
    int x, y;
};
```

The *member variable declarations* follow exactly the same syntax as ordinary variable declarations.

To use the `point` structure, a variable should be declared using type `struct point`:

```
struct Point some_point;
```

The `x` and `y` parts of this variable can now be accessed using the dot (`.`) operator. For example:

```
some_point.x = 50;
```

Open the “lab6.c” program in the editor and change the target to “lab6”. This program is a very simple demonstration of how to use structures.



Exercise 6.1: Using Simple Structures

Try building and executing the “lab6” program. Once you have run it, and tried entering some details, have a look at the source code. Make sure you understand what is happening and how it is being achieved.

Add a member variable to the structure for favourite colour. Add functionality to the program to allow the user to enter a favourite colour. The program should also display the student’s favourite colour when the other details are displayed.

6.3 Defining New Types with `typedef`

The `typedef` keyword allows you to define new types, just like `int` and `double`. The new types you define must be based on existing ones. For example:

```
typedef int Distance;
```

declares a new type called `Distance` which is identical to `int`. `typedef` statements are usually placed in the *global scope* of the program (i.e. not inside any functions) or in a header file so that the type may be used by lots of different functions. The new type may be used in a variable declaration like this:

```
Distance distance_to_tokyo;
```

If at some point in the future you wanted to make all the variables of type `Distance` more precise, you could change the `typedef` statement to define `Distance` as a `double` rather than an `int`:

```
typedef double Distance;
```

In this way `typedef` statements can help make your program more readable by making your types more flexible, and your type names more descriptive. By creating a type called `Distance` you have quite a good idea about what variables of that type are going to store.

`typedef` statements are especially useful when dealing with structures, which is why they are being introduced in this lab. When we defined a structure for holding student details it was done like this:

```
struct student_type
{
    char family_name[40];
    char given_name[40];
    int year_of_birth;
    int course_code;
};
```

This effectively defines a new type, but to use this type we have to use the `struct` keyword as well. You saw this in the “lab6” program:

```
struct student_type student;
```

We can make this shorter, and therefore easier to read (and type!), by using a `typedef`.

First, let’s change the structure definition to rename the structure type name:

```
struct long_student_type
{
    char family_name[40];
```

```
    char given_name[40];
    int year_of_birth;
    int course_code;
};
```

The prefix `long_` has been added because we are going to create a type name which can be used as shorthand, and this is the long version. We can now define a new type as shorthand for `struct long_student_type`:

```
typedef struct long_student_type student_type;
```

This new type is easier to use than the old one, because we can omit the `struct` keyword when creating variables. So:

```
student_type student;
```

declares a new variable called `student` of type `student_type`.

When using type definitions with structures it is usual to combine the `struct` definition and the `typedef` all in one statement. Like this:

```
typedef struct long_student_type
{
    char family_name[40];
    char given_name[40];
    int year_of_birth;
    int course_code;
} student_type;
```

This defines a structure `struct long_student_type` and creates a shorthand for it called `student_type`, all in one line.

Note that when using `typedef` with structures you don't need to have an "internal" structure name like `struct long_student_type` since the structure has a name due to the use of `typedef`.

```
typedef struct
{
    char family_name[40];
    char given_name[40];
    int year_of_birth;
    int course_code;
} student_type;
```

As with other type definitions, **the best place to write this code is in the global scope.**

Syntax: Type Definitions

```
typedef old_type_name new_type_name;
```

A `typedef` statement defines a new name for an existing type. The new name is an exact replacement for the old name, it can be used interchangeably with the old name (i.e. it is a *synonym*). The old type name is still valid.

For example:

```
typedef double Length;
```

Defines a new type called `Length` which is equivalent to the `double` type.

Exercise 6.2: Using typedef with Structures

Change the “lab6” program to use a `typedef` for the structure type so that the keyword `struct` does not need to be used when declaring variables.

Ensure that the type definition is in the global scope of your program (i.e. outside of any function).

Exercise 6.3: Using Your New Type Name

Create a function which displays all of the details about a student. Each piece of information should be shown on a separate line. Alter the program to use the function to display the contents of the structure in place of the final `printf` statement.

The function might have a prototype like this:

```
void display_student(student_type student);
```

6.4 Using Arrays of Structures

At the beginning of this lab the analogy was drawn between a structure and a card with details written on it. Usually there are many cards, all of the same type, kept together in a filing cabinet of some kind. So far we have only used one card. In this section we will use multiple cards by creating an array of structures.

An array of structures is easy to create by using the type that you have defined. An array of five student records would be declared like this:

```
student_type students[5];
```

You can now access individual structures in the array by specifying their index. For example:

```
students[2].year_of_birth = 1986;
```

Exercise 6.4: Using an Array of Structures

Edit the “lab6” program so that the user is first asked how many students they wish to enter details for. This could be a number anywhere between 1 and 10. If the user enters 0 the program should end. Ask the user for the details of as many students as they specified. The student details should be stored in structures in an array.

When the user has entered all of the details the program should display all of the information stored in the array. This should be achieved by calling your `display_student` function repeatedly, in a loop.

If you do not know how to attempt this, ask for help.

6.5 Graphics Option

A structure can be used to store information about a graphical object such as a line, or a shape. *Vector graphics* editing programs manipulate information about graphical objects in this way. In this lab you will store the information for a set of lines in a structure.

A line can be fully defined by:

- its start coordinate (x, y) ;
- its end coordinate (x, y) ;
- its colour (an integer value).

This information could be stored in a structure like the following:

```
typedef struct
{
    int start_x, start_y;
    int end_x, end_y;
    int colour;
} Line;
```

The program “graphics3” contains the skeleton for the program you are about to write. Change the target to “graphics3” and open the source code “graphics3.c” in the editor.



Exercise 6.5: Using Structures for Graphics

Write a program which prompts the user for the coordinates and colours of a number of lines. You might like the user to be able to state how many lines they want to use, although you will probably have to impose a maximum limit.

When the user has finished entering the information, draw the lines on the graphics window and wait for a key press.

You might like to add a feature to allow the user to edit the details of the lines that they have already entered so that they may correct mistakes.

A 2D coordinate is always made up of both x and y portions. It would make sense to define a structure to hold coordinates, or points, in a 2D space.

```
typedef struct
{
    int x, y;
} point;
```

Exercise 6.6: Nesting Structures

Add the type definition for `point_type` to your program. Change the definition of `line_type` so that the start and end location of the lines are specified as points. You will end up with a structure inside a structure.

Exercise 6.7: Implementing Simple Vector Graphics Capability

How do you think you could use one type of structure to define a range of shapes (like lines, squares, circles, triangles etc.)? What things do they have in common? What special items of information might you need for some of them? How could you make a single drawing function draw all of the shapes? Can you include fill information?

Try implementing a few of your ideas. You will need to change your structure type definition, the user input code and the drawing routine.

6.6 Music Option

In this laboratory you will use structures to create a simple sequencer. The sequencer will allow the user to input the details of various notes and then play them back, in order.

Each note can be characterised by four pieces of information:

- pitch;
- channel;
- velocity;
- duration (in milliseconds).

This information could be stored in a structure like the following:

```
typedef struct
{
    int pitch;
    int channel;
    int velocity;
    int duration;
} note;
```



The program “music3” contains the skeleton for the program you are about to write. Set the target to “music3” and open the source code.

Exercise 6.8: Using Structures for Music

Write a program which prompts the user for the details required to play a number of notes (pitch, channel velocity, duration). You might like the user to be able to state how many notes they want to use, although you will probably have to impose a maximum limit.

When the user has finished entering the information, play the notes back to the user.

You might like to add a feature to allow the user to edit the details of the notes that they have already entered so that they may correct mistakes.

Exercise 6.9: Operating on the Sequence

Alter your program so that the user may choose to play back the note sequence they have just recorded in a number of different ways:

- transposed up or down;
- in reverse order;
- the instrument(s) that is/are being used.

Exercise 6.10: A More Advanced Sequencer

The sequencer you have created is limited to playing only one note at once. Have a think about how you might lift this restriction. How would this change the information that you store in a structure? How would the playback function have to be changed? Could you incorporate changes in instrument into the sequence?

Try implementing a few of your ideas.

6.7 Audio Programming - Part 1

So far in the course you have been taught about all the fundamental and some advanced concepts in C Programming. In particular you have been taught about variables and arithmetic expressions, control flow like conditional statements and iteration loops, functions and program structure, arrays and structures. These concepts cover almost all of the C language. At this point you are well equipped to continue with audio programming and implement programs that can generate and process audio in some way. If you do not feel comfortable with the C language concepts mentioned above please revise those concepts before proceeding with this section of the lab. You should have a decent understanding of the basic topics in order to better comprehend the material that follows.

This laboratory session covers the fundamentals of audio programming. The learning outcomes of this session are the following:

- Understand the concepts of signals and digital audio signals in particular.
- Understand, at high level, how continuous-time audio signals are discretised so that they can be manipulated using a digital system.
- Become aware of Pulse-Code-Modulation for converting continuous-time signals to discrete time.
- Get familiar with the sampling rate and bit-depth of digital audio signals and understand the way these two parameters affect the quality of the signal and its size.
- Differentiate between interleaved and non-interleaved digital audio signals.
- Use the `amio.lib` library to read a digital audio file into memory (e.g. .wav file), access basic information about the file and finally write a digital audio file from memory to disk (i.e. save a file to disk).

6.7.1 Signals

The concept of signals arises in many areas of science and technology including communications, electronic circuit design, seismology, biomedical engineering, acoustics, image processing and speech pro-

cessing, just to name a few. Generally speaking, a signal conveys information about the variation / behaviour of a physical quantity / phenomenon. Mathematically, signals are functions of one or more independent variables. For example the variations in time of the voltage and current in an electrical circuit are types of signals. In this case voltage and current are the physical quantities and time is the independent variable (e.g. $v(t)$ and $i(t)$). In medicine, the electrocardiogram and the electroencephalogram are examples of signals and again the measured quantities (electrical activity of the heart and electrical activity of the scalp respectively) are functions of time. Sound (speech, music or audio in general) is also a signal where the physical quantity is acoustic pressure which varies in time. A picture is another type of signal. In this case the physical quantity is brightness which varies along two spatial variables, so a monochromatic picture is a function of two independent variables (the coordinates x and y , so the signal is $f(x, y)$). In a similar fashion we have a video signal which is a function of three independent variables (x, y coordinates and time t , i.e. $f(x, y, t)$).

We can categorise signals based on the number of the independent variables they are functions of. A signal with one independent variable is a one-dimensional signal (e.g. voltage, current, speech), with two independent variables a two-dimensional signal (e.g. photograph) so on so forth. Audio programming deals with sound signals, so in this course we are only interested in one-dimensional (1-D) signals and the independent variable will be time denoted by t .

Signals can also be categorised based on the nature of the independent variable. That is signals where the independent variable is continuous are called continuous-time signals and are defined for a continuum of values. When the independent variable is discrete then the signals are called discrete-time and are defined only for a discrete set of values. A discrete signal can be discrete by nature (e.g. demographic data, rainfall data over a period of time, stock market indices etc.) but can also arise from the sampling of continuous signals. For example sound is inherently a continuous-time signal but in order to process it using a digital computer it has to be converted into a discrete-time signal first. This conversion occurs by a process known as sampling, so a discrete-time sound signal represents successive samples of the underlying continuous-time sound signal. In this part of the course we will see how to generate, modify/process and save on disk such discrete-time audio signals.

6.7.2 Digital Audio Signals

Audio signals are signals pertaining to sound be it speech, music or any other audible sound. Sound as a physical phenomenon is simply fluctuations of pressure transmitted through a medium. The medium is usually air but sound travels through other materials as well, like water for example. Sound does need a medium to transmit though, so in outer space where there is vacuum sound cannot be transmitted. When the medium is air sound is produced by fluctuations in acoustic pressure. It is clear that sound is a continuous-time signal. It is also a one-dimensional signal since these fluctuations in acoustic pressure are a function of time only.

Analog to Digital (AD) and Digital to Analog (DA) Conversion

In order to process audio signals in the digital domain (i.e. using digital electronics, a computer etc.) sound has to be captured, digitised and stored somewhere (usually on a computer hard drive). Sound is captured by a microphone that converts acoustic pressure into an electrical signal. The microphone is a transducer, i.e. a device that converts a signal from one form of energy to another. In our case this is mechanical energy to electrical energy. At this stage the produced electrical signal (i.e. the output of the microphone) is still a continuous-time signal. This electrical signal is digitised by a process known as Analogue to Digital Conversion (ADC). An Analogue to Digital Converter (which is the device that performs analogue to digital conversion) comprises many steps but two of the most important are sampling and quantisation. The process of sampling, as mentioned in the previous sub-section, produces the discrete-time representation of a continuous-time signal; i.e. after sampling we end up with a sequence of samples that represent values of the continuous-time signal at points equally spaced in time. Sampling acts on the independent variable of the signal and produces a discrete-time signal. Quantisation on the other hand is the process that discretises the dependant variable. In the case of sound that is the electrical signal that corresponds to acoustic pressure. This process is depicted in figure 6.1.

In figure 6.1 we can see that sound emanating from the source is captured by a microphone. The captured signal is then sent into an analog to digital convertor where sampling and quantisation of the

Analog to Digital and Digital to Analog Conversion

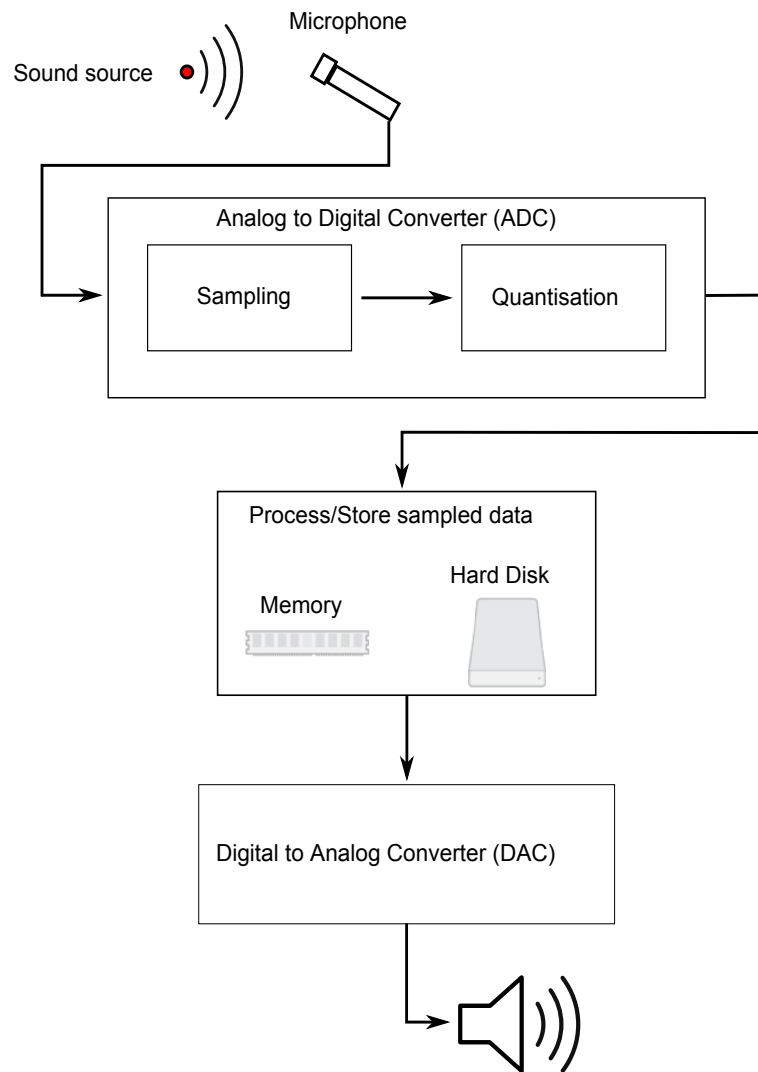


Figure 6.1: A simplified view of the Analog to Digital (AD) and Digital to Analog (DA) conversion process of audio signals.

continuous electrical signal (output of the microphone) convert it into a digital signal which can in turn be either stored into non-volatile memory (hard disks, optical disks, flash memory etc) or placed into volatile memory (e.g RAM) where it can be further processed. At this stage we have a digital representation of the analog audio signal. In many cases though we want to get back to the original analog counterpart. This is the inverse process of ADC, known as Digital to Analog Conversion (DAC). The output of the DAC can be further processed using analog signal processing (with analog electronics) and finally reproduced over loudspeakers.

Linear Pulse-Code Modulation (LPCM or PCM)

Sampling at uniform intervals and quantising a signal are parts of a method called Linear Pulse-Code Modulation (LPCM) which is the most common way to represent a continuous/analog signal in the digital domain. WAV and AIFF audio formats are two examples that contain PCM data. The following is true for PCM audio data.

Sampling Rate & Bit-Depth

It should be clear by now that sampling and quantisation affect the size and quality of the captured / digitised audio signal. We have two parameters to control the processes of sampling and quantisation namely the sampling rate (or sampling frequency or F_s) and the bit-depth (resolution) of the signal. The sampling rate is typically given in units of kilo-Hertz (kHz) or Hertz (Hz). Hertz is an SI unit and is defined as cycles per second, which in our case means how fast (at what rate) we sample the continuous-time signal. Another way to look at the sample rate is as how many samples there are in one second of audio data. That is when we see a sample rate of $44100\ Hz$ this means that one second of audio contains 44100 samples. A directly related quantity is the sampling period (T_s) which is defined mathematically as:

$$T_s = \frac{1}{F_s} \quad (6.1)$$

When quantising the signal, what we effectively want to do is to map the infinite continuous amplitude values of the analog signal into a finite discrete set of values where the number of possible values depends on the bit-depth value. The error produced by quantisation is called the quantisation error and depends on bit-depth. Bit-depth is given in units of bits per sample and is the number of bits used to represent each sample of the audio signal. A common value is 16 bits (e.g CDDA) which means that there are $2^{16} = 65536$ possible values to represent the amplitude of each sample. In digital audio processing though we usually use 32-bits to represent each sample which gives us $2^{32} = 4294967296$ possible values. Bit-depth directly affects the resolution of each sample, therefore sometimes it is also referred to as bit-resolution. A high bit-rate generally produces better quality audio since it decreases the quantisation error, improves the signal-to-noise ratio (SNR) of the reconstructed signal and provides a larger dynamic range (having said that there are techniques like dithering and over-sampling that mitigate some of these characteristics without changing the bit-depth). This will apply to audio that is already recorded using high bit rates or for audio that is generated with such values. For a thorough explanation of ADC, sampling and quantisation you can read [1][3].

Figures 6.2 until 6.5 depict the effects of sampling and quantisation on a continuous-time signal. The plots in figure 6.2 show the effect of sampling a simple sinusoid at different speeds. To be in accord with conventional notation we will denote the independent variable of an analog/continuous signal as ' t ' and will always enclose it in parenthesis (brackets) such as $x(t)$ and for digital signals we denote the independent variable as ' n ' and will always enclose it in square brackets such as $x[n]$. We can relate continuous time with discrete time using:

$$t = n \cdot T_s \text{ which leads to } x(t) = x(n \cdot T_s) \quad (6.2)$$

where $t \in \mathbb{R}$, $-\infty < t < \infty$ and $n \in \mathbb{Z}$, $-\infty < n < \infty$ with T_s as defined in equation (6.1)

Figures 6.3 until 6.5 depict the effect of quantising the amplitude of a continuous signal using different bit-depths (2 bits, 3 bits and 5 bits). NOTE that the continuous signal (blue) in all plots is of-course digital since all plots were produced from a computer generated script but for our purposes let us assume that the blue signal in all subsequent plots is continuous.

Audio Channels

An audio file on disk can comprise one (mono-channel), two (two-channel / stereo) or more channels (multi-channel). This has nothing to do with the dimensionality of the underlying signals. That is a mono-channel audio file comprises one 1-D signal, a two-channel audio file comprises two 1-D signals, so on so forth. One question that comes to mind is how audio samples are grouped inside an audio file that might have a multi-channel format. For Linear Pulse-Code Modulated signals (LPCM as explained in previous sections) we can have a non-interleaved or an interleaved configuration. In the non-interleaved case the samples of each channel are grouped separately meaning that samples belonging to each channel are stored in separate contiguous memory locations. In the interleaved configuration the samples of each channel are interleaved, meaning that the audio samples of each channel alternate in a single stream. These two configurations are depicted in figure 6.6. In both configurations the signal comprises a number of frames, where each frame comprises M channel audio samples. In case of a

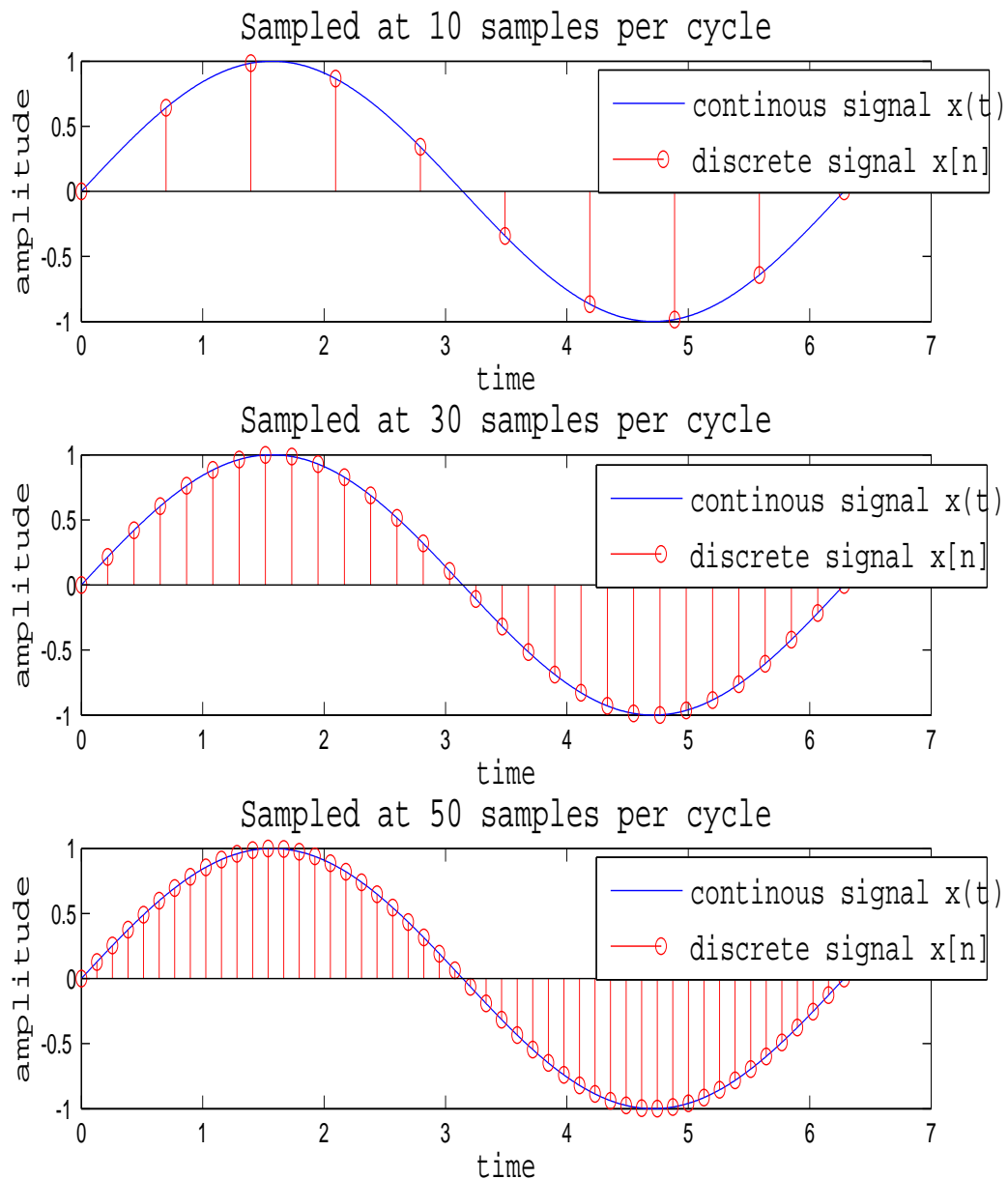


Figure 6.2: Demonstration of sampling a simple sinusoid at different rates. At high sample rates the digital signal is a better representation since we have more information (more samples) of the original signal.

mono-channel file $M=1$ so a frame comprises one sample. In a stereo case, $M=2$ so each frame comprises two samples (left channel sample followed by right channel sample) so on so forth.

Interleaved audio data is a very common configuration and we are going to use it throughout the exercises. In `amio_lib.h` library, the function that is responsible for loading audio data into memory produces an interleaved buffer of audio data and we should take that into consideration especially when processing multi-channel audio files.

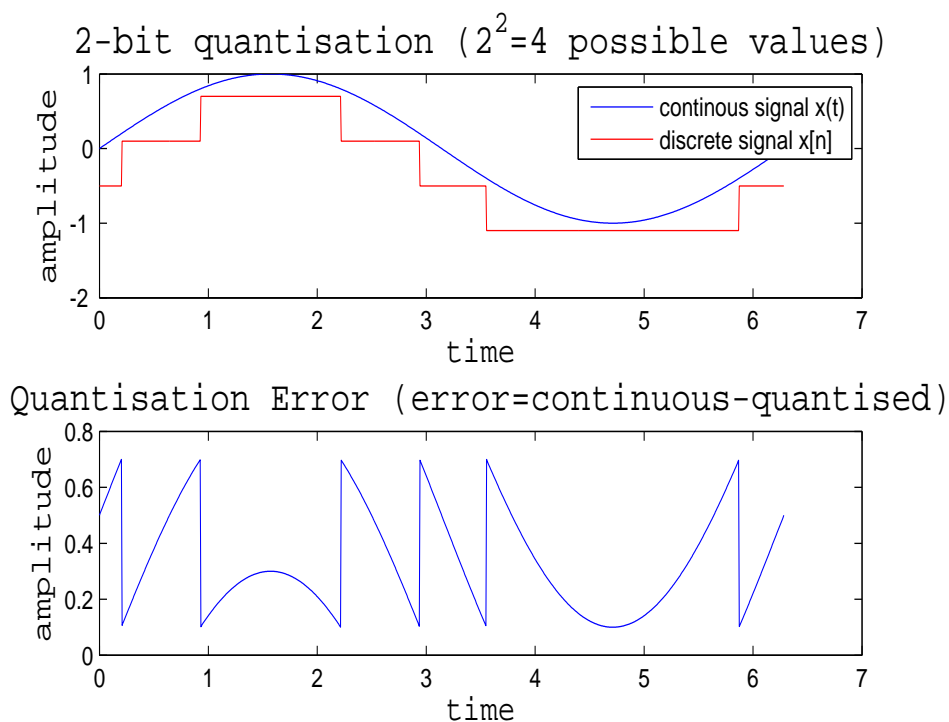


Figure 6.3: 2-bit quantisation. Each audio sample can take a value out of only four possible values.

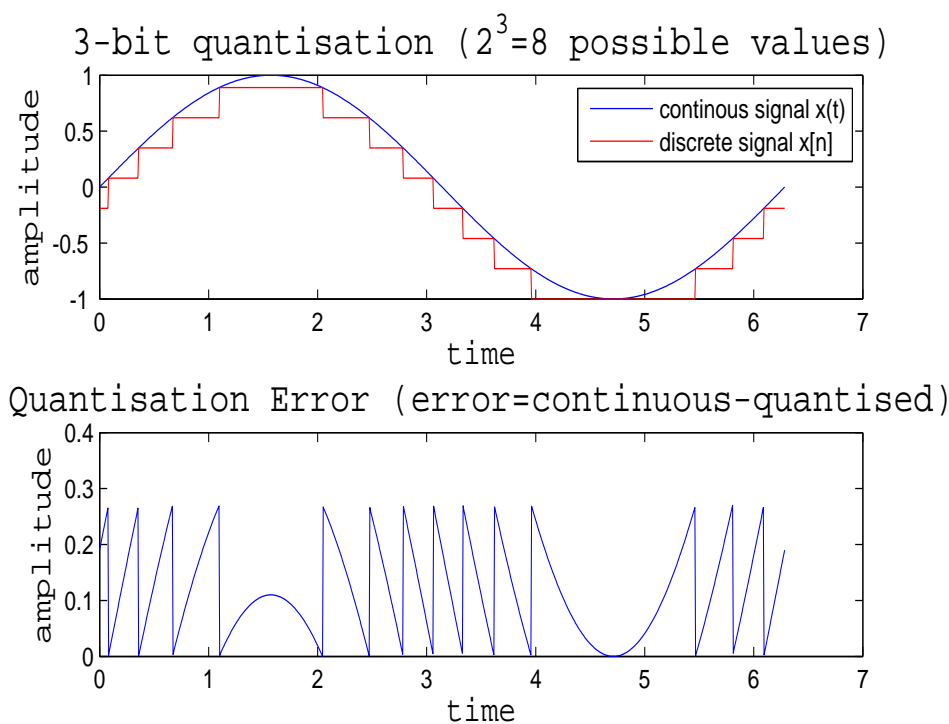


Figure 6.4: 3-bit quantisation gives eight values to choose from. We can see that the quantisation error decreased.

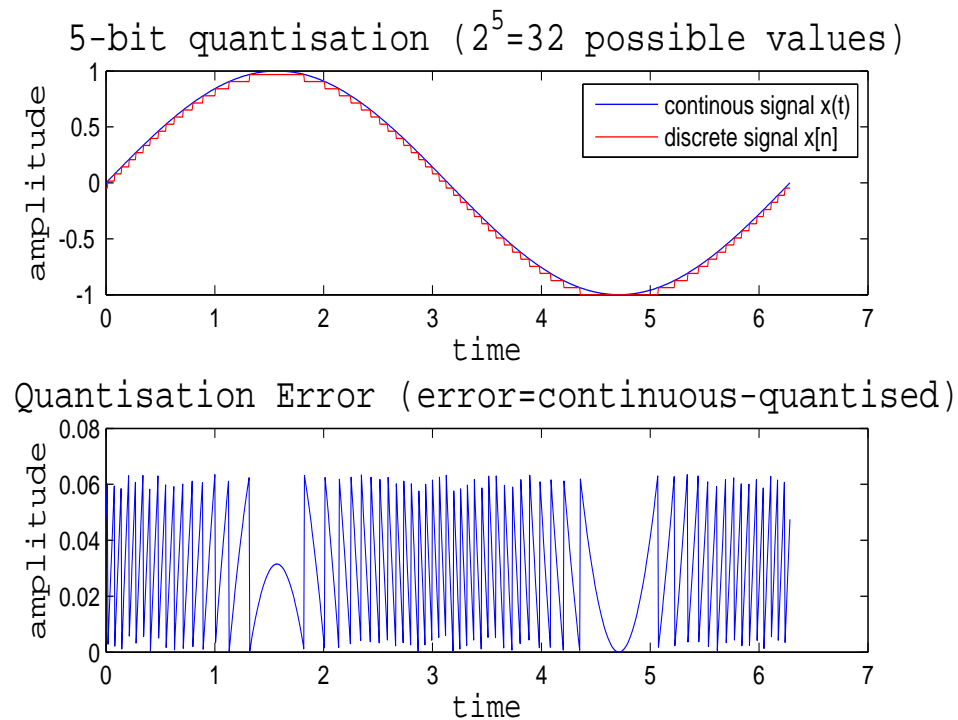


Figure 6.5: With five bits we get 32 possible values to choose from and the quantisation error decreased dramatically compared with the 2-bit quantiser. In the top plot we can also see how the digital signal (red) begins to look similar to the original (blue).

6.7.3 Audio & MIDI Input/Output Library (`amio_lib`)

`amio_lib` is a library specifically build for this course. It comprises three third party libraries, namely `libsndfile`, `portaudio` and `portmidi`. `libsndfile` provides functions that deal with reading / writing audio files from / to disk. `portaudio` provides real-time audio capabilities and `portmidi` provides MIDI functionality. We chose these libraries for all audio and MIDI needs of the course for three main reasons:

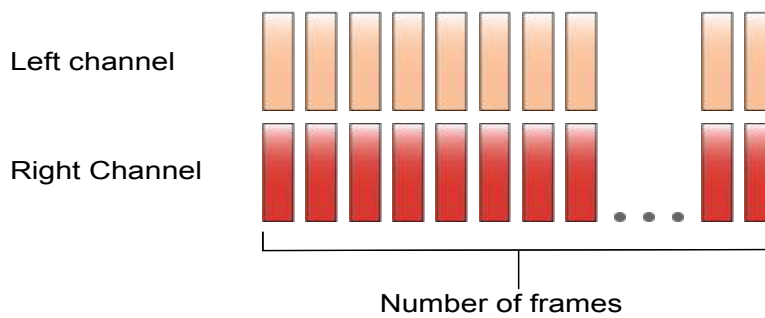
1. They are written in pure C and are well suited for this course.
2. They are cross-platform libraries meaning that you can write your code once and compile (the same code) in multiple architectures like for example Windows, Apple OS and Linux.
3. They are free software which means that we can use them for educational purposes without having to pay any license fee.

`amio_lib` combines the power and functionality of all aforementioned libraries providing a single, expandable, simple and easy to learn Application Programming Interface (API) suitable for beginners in the areas of audio and MIDI processing. Of course `amio_lib` does not provide the full functionality of `libsndfile`, `portaudio` and `portmidi` but definitely enough for this course. All examples and exercises in the next section use `amio_lib`. To get the `amio_lib` functionality in your programs all you have to do is add `#include "amio_lib.h"` at the top of your implementation file.

The program “audio01” contains code which loads a wav file into memory, displays some basic information about the file and saves the file unaltered on disk. Set the target to “audio01” and open the source code.



a) Non-interleaved LPCM Stereo Audio



b) Interleaved LPCM Stereo Audio

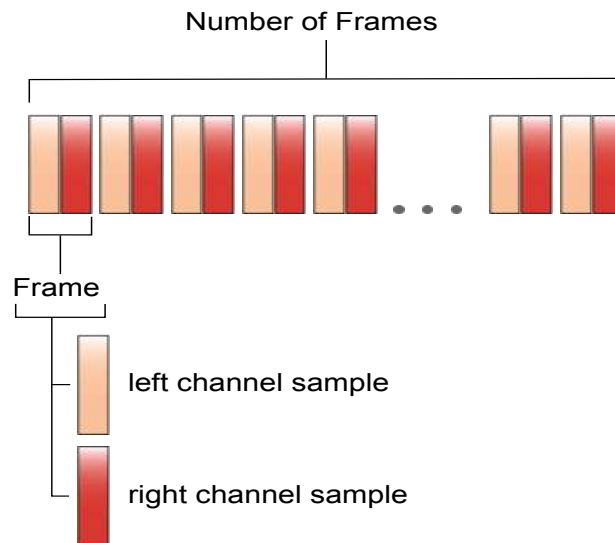


Figure 6.6: Figure a) depicts how LPCM stereo audio data is stored in a non-interleaved way. Left channel samples are grouped separately from right channel samples. Figure b) depicts LPCM stereo audio data interleaved. The audio stream consists of frames, where each frame comprises M channels samples (for stereo, $M=2$ thus each frame consists of 2 samples).

Exercise 6.11: Reading an audio file into memory and writing it back to disk

This example takes you through the processes of reading an audio file into memory and writing it back to disk, unaltered but with a different name and on a different physical file, something like a rename program would do. This example introduces the two functions that allow us to perform such operations namely, `wavread` and `wavwrite` (the naming of the functions was taken by the similar functions found in MATLAB software package).

Open file `audio01.c` into CodeBlocks (or Xcode) and read through it. There are plenty of comments that explain what each step does. First we create two `SIGNAL` structures to hold information about the input and output audio files (sampling rate, bit-depth, format, number of channels, number of frames and of course the actual data). Then we use the `wavread` function to read the file into memory. `wavread` actually performs a series of steps in the background; first of all it opens the audio file we requested, then it loads its data into memory and finally it closes the file. This way the original file remains unaltered, as it should be. We should not be concerned about these steps because `wavread` takes care of everything.

Your exercise is to read through the code in `audio01.c` carefully and make sure you understand what it does. If in doubt ask one of the demonstrators.

A few words on error handling

At this point note what `wavread` returns. The return value is an integer which represents an error code. When creating Application Programming Interfaces, especially in C, it is a usual design pattern for the functions to return an error code, especially functions that perform critical and/or complicated operations. As programmers we should always check for errors returned by functions and act accordingly. This is what the following lines of code do:

```
err=wavread(kInputSignalLocation, &input); checkErr(err,kSndFileSystem,"Failed
to read audio file into memory"); amio_lib uses three different libraries in the background
each of which has its own error reporting system (portaudio and portmidi error reporting mecha-
nisms are very similar).
```

`checkErr` is an utility function which provides a single error reporting interface. For more information on that and other functions read the comments included in `amio_lib.h`.

After reading the file into memory the program proceeds by displaying information about the file using another utility function `displaySndInfo`.

After displaying the info to the user the program uses yet another utility function, `Fill_SIGNAL` which prepares the output `SIGNAL` structure. By preparing we mean that the fields of the `SIGNAL` structure are populated with values. This function is a helper function and you could also populate the fields of the `SIGNAL` structure in the usual way of accessing each field of the structure and assigning a value to it. Having a `SIGNAL` structure ready for output the only thing that is left is to actually assign the audio data to write to disk and this is achieved with the following line:

```
output.data=input.data;
```

Recall that 'input' and 'output' are both `SIGNAL` structures and each has a `data` field which is a pointer to a memory location where the actual sample data reside. In this case we merely copy the pointer of the input data to the pointer of the output data so that `output.data` pointer points to `input.data`.

Finally the program uses the `wavwrite` function to write the output `SIGNAL` into a file on disk. It should be clear that in this example the creation of an output `SIGNAL` was not necessary; we could have simply passed `SIGNAL` input to the `wavread` function. Make sure you understand the logic behind this statement and the program in general. If you have any questions please ask one of the demonstrators.

6.8 Summary

This lab has introduced you to structures, a powerful way of collecting together information. An array collects information, but a structure is often useful because:

- the different parts of it (*members*) may have different types;
- the members are named, rather than simply referred to by index.

The middle part of the lab introduced type definitions, and showed a way in which they are commonly used when defining structures.

You should also have used a collection of structures in an array, both in a simple example for storing student details and in the graphics and music options. The graphics option used an array of structures to deal with graphical objects such as lines in a similar way to vector graphics editing programs. The music option used an array of structures as the basis for a simple sequencer.

