Praat under the hood

David Weenink
Praat team

For technically interested readers some of the programming aspects of the Praat program will be discussed. We explain how reading and writing of objects takes place by explaining how Paul Boersma used the macro programming facilities of the C++ language. We further explain the inclusion of the external programs, especially the eSpeak NG speech synthesizer. To entertain the not technically interested reader we have included some specially crafted personalised sudokus that can be solved.

1 Introduction

In this article I will highlight some of the internals of the computer program Praat (Boersma, Weenink, & Shchupak, 2026) which are not publicly known because this knowledge is not necessary for using the program and it also cannot be found in the Praat manual pages. This under-the-hood information can of course be distilled from the Praat source code. If you want to do this distillation process yourself then, besides detailed knowledge of the C++ programming language and its macro language, information about the structure of the Praat code base will also be necessary. In this article I will try to explain some of these internals to the non-programming outside world without presupposing this specific detailed programming knowledge, although it helps a lot if you do have programming experience. I do suppose that the reader is familiar with using the Praat program. This means that the user knows that after starting Praat two windows appear on the screen of your computer: the Object window with the title Praat Objects on top and the Picture window with the title Praat Picture on top. Both windows also have a fixed menu on top; the menu of the Object window are Praat, New, Open, Save and Help. Parts of the Object window are the list of objects and the dynamic menu whose items depend on the selected object(s) in this list of objects.

The world of Praat consists of objects which we call *Things*. Some Things are special, they can be saved on, or read from, a storage medium. These Things are called *Data*. Many different types of Data objects can be handled by Praat and some common examples are a Sound, a Spectrogram and a Pitch or more exotic ones like a Permutation, an Excitation or a PCA. The Data objects that Praat normally manipulates are visible in Praat's list of objects. Thing objects,

Linguistics in Amsterdam, 2026: 215–226
© David Weenink

on the other hand, don't need to be saved because the Praat program knows how to create them on demand and Thing objects will therefore never be visible in the list of objects. Editors are examples of Things. A SoundEditor manipulates a Data object of type Sound; the Sound object appears in the list of objects but the SoundEditor does not. After the manipulation the Sound Data object may be saved on a storage medium and the SoundEditor object can be destroyed because it can be simply recreated by Praat. Once a Sound object is visible in the list of objects a new SoundEditor will be created if you click the View & Edit command of the Sound's dynamic menu. The current 7.0.02 version of Praat uses 162 different readable Data types that could be listed in the Info window by using the command Praat>Technical>List readable types of objects. This notation means that you can find the command in the Technical sub menu of the Praat menu in Praat.

In the following section 2 we describe how Praat reads and writes its data; the examples will show some advanced usages of macros. It will be followed by section 3 where we show how the external program eSpeak NG was incorporated in the Praat program to deliver text-to-speech synthesis.

2 How does Praat know how to read Data?

Computer files are of two types: text files and binary files. Text files contain text that you can read and binary files if opened in a text editor in general appears as unreadable gibberish. Data often times needs less storage space if it is stored in a binary file.

For a computer program to read a data file, the program needs to know exactly what type of data it is and how the data is stored in the file. This information is called the file format. In former times the file format would be hinted at only by a filename extension like '`.wav`' or '`.mp3`' and the computer program would not read a file if the extension was not right. A filename extension has never been a reliable and robust way to indicate the type of a file because the extension is part of the filename and the filename can be changed at will by the owner of the file thereby masking the file's content.

A much safer way is to store this information not in the file's name but in the file's content as a kind of unique signature. In this way a file extension merely serves as a hint for the file's content. For example the signature of an audio file in WAV-format is in the first 12 bytes of the binary file and reads '`RIFFxxxxWAVE`'. This means that if the first four bytes of the file are the four characters '`RIFF`', followed by four bytes that are not important in this context, followed with four bytes that read '`WAVE`' the file is probably a Microsoft audio file. The WAV-format is free, well documented and relatively simple to implement which means that Praat knows how to read this type of audio file. For

many other free sound file formats which are considerably more complicated like FLAC (Free Lossless Audio Codec) and MP3 we use implemented external libraries to read these kinds of files. Other non-free audio file formats like AAC (Advanced Audio Codec) restrict usage, modification or distribution through software patents and licensing fees and will therefore not be implemented. Despite the many external audio representations like WAV, FLAC, MP3, and many more, in Praat they are all represented as a Sound object.

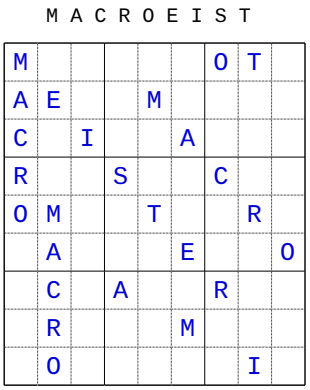
The more than one hundred and fifty data types that Praat internally uses can all be saved to a file and read from file by Praat because of a nifty set of macros that I will roughly describe here. These macros have been introduced by Paul Boersma when he started working on Praat more than thirty years ago and I hope to show that they besides saving the programmer a lot of time are also very elegant. This time saving occurs because by using these macros the reading and writing code will be generated automatically. So what is, in general, important for data type storage and retrieval? To read data from storage we need to know what all the different parts of the data are and where they are stored in the file. The most economic strategy to guarantee this, is to read the data from storage in the same order as this data was written. This has the consequence that the reading and writing code exactly have to know the specifics of the data type they are processing. In other words: each data type needs its own specific reading and writing code.

Before we are going to show the simplified macros let's first explain what macros are. A macro is an automated instruction that tells the compiler to replace a specific sequence of input characters with a preset output sequence. Macros allow developers to write shortcuts or reuse code. A macro has a *name* and a *value* and they act like text-replacement where the name in the source code is replaced by the value. There are almost no limitations on the name and the value of a macro, although, to distinguish a macro name from a variable we write all macro names with uppercase characters. The macro's value can range from a single item, like a number, to many lines of computer code. The examples that we will show in the sequel are used for "code generation" and only the parts needed are shown. Macros are processed, in a so called macro expansion step, before the actual translation (compilation) of the source code to machine code happens.

Let's show what kind of macros we will use to define a new type of data. Just as a simple example suppose we want to define a new data type called *ShortSound*. The ShortSound needs as minimal attributes only a sampling frequency and a vector to contain the sample values. We will show that by using some specially constructed macro definitions we can read and write the new data type with minimal effort: we should not have to write any new computer code for reading and writing but simply use some predefined macros for accomplishing

that.

Besides being just swapping the name by its value, macros also can have arguments and this fact will be used extensively as we will show in the sequel.



Listing 1: The definition of the ShortSound data type.

```
1 DEFINE_CLASS (ShortSound, Data)
2   REAL (samplingFrequency)
3   INTEGER (numberOfSamples)
4   INTVECTOR (samples, numberOfSamples)
5 END_DEFINE_CLASS
```

The text in listing 1 which should be saved in a file with the name 'ShortSound_def.h' *defines* the data type ShortSound by using five different macros. The first line starts with a DEFINE_CLASS macro with two arguments that we interpret as defining the new ShortSound type as inheriting from the Data class. Inheriting means that methods that are valid for an object of type Data also work for a ShortSound. The next line shows a macro with one argument: REAL (samplingFrequency). We interpret it here as the parameter samplingFrequency is of type real number. The next line with INTEGER (numberOfSamples) can in an analogous manner be interpreted: the parameter numberOfSamples is of type integer number. The line INTVECTOR (samples, numberOfSamples) then means that samples is an array vector that can store numberOfSamples integer values. The line with END_DEFINE_CLASS concludes this type definition. This interpretation of the macros is just one possible way. We show how the macros in this file can be expanded in different ways to automatically generate code to read the object from a file and write the object to a file.

Listing 2 shows how we should use this definition file in an actual piece of code. A line like #include "oo_WRITE_TEXT.h" means that the content of file oo_WRITE_TEXT.h has to be included at that particular position in the file ShortSound.cpp. These first four lines therefore include three different files into this file (and yes, the content of file ShortSound_def.h is included twice). The relevant content of the predefined files oo_WRITE_TEXT.h and oo_READ_TEXT.h is shown in listings 3 and 4, respectively. The effect of the inclusion of these two files is that the five macros in listing 1 will be expanded in the two different ways defined by the macro definitions in these two files. The result of the expansions is shown in listing 5. These two files will be used for all the data types that are defined in Praat and they redefine the macros used in

R E A D W I T X !

	R		I		!			X
		E					R	
		W	A				E	
T			R	D			A	
	E			I			D	
D		X			T	R		
R			T			E		
				!		A		I
E						D		

Listing 2: First lines of `ShortSound.cpp`.

```
#include "oo_WRITE_TEXT.h"
#include "ShortSound_def.h"
#include "oo_READ_TEXT.h"
#include "ShortSound_def.h"
...
...
```

any defined type like that in listing 1[1]. In listing 3 the `DEFINE_CLASS` macro is (re)defined to produce a function that can write to a file. The first two lines of example 5 show the result of the macro expansion of the first line in listing 1 with the expansion defined in the first three lines of 3: for the parameter `_newClass` the value `ShortSound` is substituted and for `_parentClass` the value `Data`. In line two of listing 1, the macro definition `REAL(samplingFrequency)` is changed to `writeRealAsText (file, samplingFrequency)`. The other macro expansions in listing 1 go in an analogous way. In this way the redefinition of the same macros in `oo_READ_TEXT.h` define the `ShortSound::readText (File file)` method. The expansions of the text in example 2 with the redefinitions in listings 3 and 4 result in the code shown in listing 5.

Listing 3: File `oo_WRITE_TEXT.h` redefines the macros used in listing 1. The backslash '\' at the end of a line indicates that the next line belongs to the same macro. The use of the `oo_READING` and `oo_WRITING` macros is explained in section 2.1 and section 2.2 explains the `oo_DECLARING` macro.

```
#define DEFINE_CLASS(_newClass,_parentClass) \
  _newClass :: writeText (File file) {\
  _parentClass :: writeText (file);
#define REAL(_x) writeRealAsText (file, _x)
#define INTEGER(_x) writeIntegerAsText (file, _x)
#define INTVECTOR(_x,_n) writeIntVectorAsText (file, _x, _n)
#define END_DEFINE_CLASS }

#define oo_READING 0
#define oo_WRITING 1
#define oo_DECLARING 0
```

The story is not complete yet because up till now we have only shown how the read and write methods for a ShortSound object can be generated automatically if we *know* we are dealing with a ShortSound object. But how do we know that? We still have to show how Praat handles file signatures.

[1] Each `oo_*.h` file first makes the preprocessor forget the previous definitions.

Listing 4: The file `oo_READ_TEXT.h` redefines the macros used in listing 1.

```
#define DEFINE_CLASS(_newClass,_parentClass) \
  _newClass :: readText (File file) {\
  _parentClass :: readText (file);
#define REAL(_x) readRealFromText (file, _x)
#define INTEGER(_x) readIntegerFromText (file, _x)
#define INTVECTOR(_x,_n) readIntVectorFromText (file, _x, _n)
#define END_DEFINE_CLASS }

#define oo_READING 1
#define oo_WRITING 0
#define oo_DECLARING 0
```

Listing 5: After the expansion of the macros in `oo_READ_TEXT.h` and `oo_WRITE_TEXT.h` the text in ShortSound.cpp becomes the following.

```
void ShortSound :: writeText (File file) {
  Data :: writeText (file);
  writeRealAsText (file, samplingFrequency);
  writeIntegerAsText (file, numberOfSamples);
  writeIntVectorAsText (file, samples, numberOfSamples);
}

void ShortSound :: readText (File file) {
  Data :: readText (file);
  readRealFromText (file, samplingFrequency);
  readIntegerFromText (file, numberOfSamples);
  readIntVectorFromText (file, samples, numberOfSamples);
}
```

If we had used the `Save as text file...` command to write the ShortSound object to a file named `shortSound.Sound` and then had opened this file in a text editor or in Praat's ScriptEditor it could have looked as in listing 6. The first two lines in this file serve as the file's signature. After having read the first line of the file, Praat's file recognizer immediately knows it is dealing with a text file that has been created by writing one of Praat's objects to file: all Data objects in Praat when saved as a text file have `ooTextFile` in the first line of the file. The type of object, a ShortSound, is revealed in the second line. Praat now knows that the data that follow belong to a ShortSound. The next step will be that Praat creates a rudimentary ShortSound object. Rudimentary because it does not know the details of this object because they have not been read from disk yet. Once the ShortSound is created the reading of the file is dispatched to the `ShortSound::readText` method as was defined in the lower part of listing 5. Praat then starts reading the data that has been saved in the file by the parent object. Because the parent didn't save anything, nothing will be read by `Data::readText`. Next the real number 44100.0 is read and assigned to the ShortSound's `samplingFrequency` variable. This is followed by reading the

integer `numberOfSamples` value, 100, from the file and assigning it to the corresponding variable in the ShortSound object. The `readIntVectorFromText` has to do several things. First it has to create a vector of `numberOfSamples` integer elements and then successively read the values from the file. Once this is completed the object is complete and it will appear in the list of objects.

The mechanism explained above will be used for all data files that have a specific signature that Praat is aware of. This has the effect that for all the files with a signature we only need one `Read from file...` command.

Listing 6: Content of the `ShortSound.Sound` file.

```
1 File type = "ooTextFile"
2 Object class = "ShortSound"
3 samplingFrequency = 44100.0
4 numberOfSamples = 100
5 samples [ ]:
6 samples [1] = -1
7 samples [2] = 32000
8 <etc... samples [3..99]>
9 samples [100] = 1
```

S H O R T U N D Z

	S	H	O	R	T		Z	
		O						S
R			U					H
		Z		N				O
					D			R
S								T
		S	H	O	R	T		
H	Z						O	
			N		U			

2.1 How can we make a change to a data type?

Having worked with ShortSound objects for a time we find it inconvenient that the samples are in integer format, we would prefer working with a version that uses real numbers because all sound processing happens in the real number domain. The consequence would be that in the ShortSound definition file of listing 1 we need to use `REALVECTOR` instead of `INTVECTOR`. This poses no problem for representing the object in Praat's memory or for writing to a file, however, we would then have two file versions at the same time that can not be distinguished from one another: the older version with integer sample values and the new version with real sample values. For a text file, reading would probably not be a problem, however, for a binary file it would cause a serious problem because real numbers need more file space than integer numbers and the interpretation of the binary data differs. The solution to this problem is that we also need to write the data type version to the file to make our file reading *version-aware*. The default version number is 0, unless otherwise indicated and this 0 is *not* marked in the file. If we would change the version number to 1 for our ShortSound object, the second line of the file would change to `Object class = "ShortSound 1"`. The definition file `ShortSound_def.h` would also have to change as listing 7 shows. This listing needs some explanation. In line 4 we are testing whether the macro `oo_READING` is defined. This macro has the value 1 when we are reading and 0 otherwise as can be checked from listings 1, 3 and 4. Therefore, if we are

not reading only line 15 after the `#else` survives, If we are reading, the version read from the file decides at run time between lines 6–10 (old file) and line 12 (newer files). Only when the version number equals 0 we have a special case: for version 0 we need to read the older format that has the integer vector. This is done by the expansion of the `INTVECTOR` macro where the sample values are read into a temporary integer vector called `tmp`. The next lines are code, that first create a real vector of the right size and then converts the integer values `tmp[i]` to real values `samples[i]`. The conversion is implicit by assigning the integer value to the real variable. So, after all, we have to do a little bit of programming to make the reading of the old version work. Because the data type version has to be known to the reading methods of the ShortSound the expansion of the `DEFINE_CLASS` macro has to change. The `ShortSound::readText` needs the version read from the file as an extra argument and therefore is changed to `ShortSound::readText (File file, integer formatVersion)` (in general the read methods of all the classes need this additional argument to cope with different versions). In this way the version from the file can be compared to the value supplied to the `oo_VERSION_UNTIL` macro.

Listing 7: The new `ShortSound_def.h` file with version control.

```
 1 DEFINE_CLASS (ShortSound, Data)
 2   REAL (samplingFrequency)
 3   INTEGER (numberOfSamples)
 4   #if oo_READING
 5     oo_VERSION_UNTIL (1)
 6       INTVECTOR (tmp, numberOfSamples)
 7       samples = createRealVector (numberOfSamples)
 8       for i to numberOfSamples {
 9         samples [i] = tmp [i];
10       }
11     oo_VERSION_ELSE
12       REALVECTOR (samples, numberOfSamples)
13     oo_VERSION_END
14   #else
15     REALVECTOR (samples, numberOfSamples)
16   #endif
17 END_DEFINE_CLASS
```

2.2 The declaration part

Having explained the handling of different versions of a data type, we are ready for the `oo_DECLARING` macro which declares the type. Just like for the reading and writing macros this happens in a separate file, named `oo.h`, and the relevant parts for our ShortSound type are shown in listing 8.

The `DEFINE_CLASS` macro opens the definition of the class.[2] The macro

[2] In C++ there is no big difference in using `struct` or `class` for the definition of a class. The major difference is that members of a struct are default *public* and for a class they are *private*.

Listing 8: The file `oo.h` redefines the macros used in listing 1.

```
#define DEFINE_CLASS(_newClass,_parentClass) \
  struct _newClass : public _parentClass {

#define REAL(_x) double x;
#define INTEGER(_x) int32 _x;
#define INTVECTOR(_x,_n) vector<int32> _x;
#define REALVECTOR(_x,_n) vector<double> _x;

#define END_DEFINE_CLASS \
  void writeText (File file) override; \
  void readText (File file, int formatVersion); \
  };

#define oo_READING 0
#define oo_WRITING 0
#define oo_DECLARING 1
```

`REAL(_x)` is simply expanded to the declaration `double _x;` while the expansion of the macro `INTEGER(_x)` results in a `int32 _x;` declaration. The `END_DEFINE_CLASS` macro declares the `readText` and `writeText` methods for the class. This results in the expansion shown in listing 9.

Listing 9: The type declaration of the ShortSound as defined in 7.

```
struct ShortSound : public Data {
  double samplingFrequency;
  int32 numberOfSamples;
  vector<double> samples;

  void void writeText (File file) override;
  void readText (File file, int formatVersion);
};
```

To make this work for all data types defined in Praat on all platforms several additional things need to be in place:

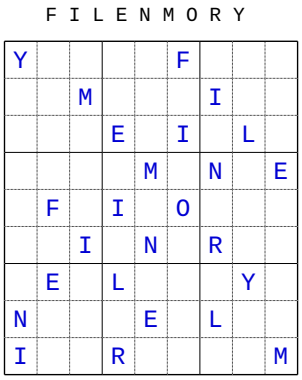
- The data type definition file, in our case `ShortSound_def.h`, needs to be included in the corresponding data type header file `ShortSound.h`.
- The declaration file `oo.h` needs to be included in the header file also, before the included data `*_def.h` file, because the declaration macros need to be defined before they can be expanded.
- The data files need to be stored in a fixed format independent of the computer hardware. This permits exchanging files between computer platforms without hassle.

This concludes the section on the ingenious use of macros, i.e. the *one file many products* idea, for automatically reading and writing data objects to text files. We mention that we have highlighted only the basics of this automation process and tried to hide the complexities. The same definition file `ShortSound_def.h` can also serve for reading and writing to binary files. Binary files get the `ooBinaryFile` signature. Different expansions of this definition file can also be used for *comparing*, *inspection* and *destroying* of data objects.[3]

3 Text to speech synthesis with eSpeak NG

For the implementation of text-to-speech synthesis we decided in December 2015 not to write the software ourselves but to use the computer code from a program called `eSpeak`. It was developed by Jonathan Duddington until 2015. Then Reece Dun cloned it and together with a group of developers they maintain and actualize it under the name `eSpeak NG`. We preferred this program because the source code was publicly available at https://github.com/espeak-ng/espeak-ng and synthesis rules for more than fifty languages were already present. However, the adaptation of `eSpeak NG` as the text-to-speech engine for Praat was not a trivial case. The current 1.52–dev version implemented in Praat is from 2024–08–24. It supports 137 languages with 104 voices. The data for all these languages and voices must be stored in a special `espeak-data` directory on the hard disk of the computer. The language data consists of two parts: the rules to convert text input to phoneme codes and a dictionary with words that do not follow these rules. All these data files have to be supplied together with the `eSpeak NG` text-to-speech synthesizer to generate speech. However, Praat is distributed as one simple executable file and one would like to keep it that simple. The implication of incorporating `eSpeak NG` would be that instead of downloading only one executable program namely Praat, the `espeak-data` language and voices files also have to be supplied. These `espeak-data` files would have to be at a fixed position in the user's file hierarchy in order for the synthesizer to be able to read this data. The simplicity of downloading only the Praat program would be lost because now the Praat installer also needs access to the user's file system to guarantee that the data files are located where Praat can find them. In another scenario the Praat website would supply the `espeak-data` and the user could download the data to a fixed location. Version differences between the `eSpeak NG` implemented in Praat and the language and voice files on the user's side could arise if newer versions of the synthesizer were implemented in Praat and the user forgets to download the new data files. The only safe way

[3] The complete macro files can be found in the `sys` directory of the Praat code. The `oo.h` defines the macros in a C++ header type way. The `oo_*.h` files (re)define them to support other methods. Data definition files can be present in all Praat source directories and have names that end with `_def.h`. An example is `Pitch_def.h`. You will also notice that the inheritance is more detailed than I have assumed here.



Listing 10: The core of the FileInMemory data structure.

```
1 DEFINE_CLASS (FileInMemory, Data)
2   STRING (filePath)
3   INTEGER (numberOfBytes)
4   INTEGER (position)
5   BYTEVECTOR (data, numberOfBytes)
6   <internal state variables>
7 END_DEFINE_CLASS
```

to circumvent these obstacles is to always distribute the language and voices files together within the Praat executable. The consequence of this is that, first of all, the data that normally is stored in the language and voices files on the hard disk is now stored within the memory of the Praat executable, and, in the second place, that all the file accesses of the speech synthesizer need to be translated to memory accesses. Luckily the total file size of the language and voices files is only a couple of megabytes which makes this task feasible. However, this involved quite a lot of work to implement the synthesizer in a platform independent way. We started by defining a FileInMemory data structure which enables storing of a file in the memory of the computer. The essential part of the FileInMemory class is abstracted in listing 10. Each of the languages and voices files can then be represented by one FileInMemory object. The file data have to be copied into the byte vector `data` part while its file path is copied into the `filepath` string variable. The `filePath` variable makes them identifiable. To make the data in memory behave like a normal file we also had to provide the standard file I/O functions like fopen, fclose, fread and others that are used within the `eSpeak NG` synthesizer part.[4] All the FileInMemory objects will be collected in a special FileInMemorySet. To actually make all the necessary data available to the synthesizer the following things to be done.

- Write code that can create a FileInMemory object from a file. This means opening the file, copy its content into the `data` buffer and its file name into the `filePath` variable and set the internal state variables.
- Write code that uses the method above to convert all the files in a directory and its subdirectories to FileInMemory objects and collects these objects in a FileInMemorySet. This set can then be searched for any desired language or voices (file) name.
- Write code that can output a FileInMemorySet as a C++ code file. This

[4] On Linux `fmemopen` is a function that opens a stream in memory and it was available at that time, for Windows and MacIntosh there was no alternative.

is the necessary bootstrap for including the data into the Praat executable; the resulting file can then be compiled and because it contains all the necessary data the `Espeak` synthesizer data are now within the executable.

- Change the `Espeak` code in such a way that all its input is directed to the included FileInMemorySet and not to files outside the Praat executable.

The still more detailed explanation of the work involved to do this yourself can be found in the `generate/espeak/GENERATE.praat` script from the Praat sources at https://github.com/praat/praat.github.io.

4 Conclusion

Some technical aspects of creating the Praat computer code have been highlighted. We finish with the Praat sudoku.

P A U L D V I R T

P	A	U	L			I		R
					U		P	
							A	
	D		P				U	
V		A		R			L	
			V		A			P
		R		I		A		
	U				D		T	
		P	A	U	L			

References

Boersma, P., Weenink, D., & Shchupak, A. (2026). Praat: Doing phonetics by computer [computer program], version 7.0.02. http://www.praat.org/

Duddington, J., & Dunn, R. (2024). Espeak-ng (version 1.52-dev). https://github.com/espeak-ng/espeak-ng

David Weenink
`dweenink@xs4all.nl`