Functions

Let's start again with what the NSIS Help file says on them:

"Functions are similar to Sections in that they contain zero or more instructions."

Let's try a different definition. Functions are like macros, except that the code does -not- actually get inserted or copy/pasted, if you will, when compiling. The compiled code only exists just once in your final installer.

passing parameters/arguments to Functions is typically done through variables...

Example 1.2.5

```
var myVar

Function Hello
  DetailPrint "Hello $myVar"
FunctionEnd

Section Test
  StrCpy $myVar "World"
  Call Hello
SectionEnd
```

... or, more commonly, the stack (to save memory used by variables)

Example 1.2.6 SectionEnd </highlight-nsis>

Which ends up acting as (but please note that the code is not actually inserted as such!):

Example 1.2.7

```
Section Test
  Push "World"
  ; At this point, the stack looks like the following (top is left, bottom is
right)
  ; :: "World" -rest of the stack, if anything-
  Exch $0
  ; At this point the old value of $0 is now at the top of the stack, while
"World" is copied into variable $0
  ; :: old$0 -rest of the stack, if anything-
  DetailPrint "Hello World"
  Pop $0
  ; At this point the old value of $0 is restored, and the stack just looks
like the following:
  ; :: -rest of the stack, if anything-
SectionEnd
```

Functions can either clear the stack relevant to the function, as in the above, or they can leave an item on the stack for use after the function.

Example 1.2.8

```
Function isDir
  Exch $0
  IfFileExists "$0\*.*" _dir _notdir
  _dir:
    StrCpy $0 "true"
    return
  _notdir:
    StrCpy $0 "false"
FunctionEnd

Section Test
  Push "some path on your drive"
  Call isDir
  Pop $0
SectionEnd
```

Where in the above code, after the call to "isDir", `$0` contains "true" if the path was a directory, and "false" if not.

Macros

From the NSIS help file: "Macros are used to insert code at compile time, depending on defines and using the values of the defines. The macro's commands are inserted at compile time. This allows you to write general code only once and use it a lot of times but with few changes."

If you need to use macros please make sure you create a nsh file and it's saved into your `Include` folder. Your script has to include the nsh file where the macro is defined. Of course, please make sure output file is defined. Please see example

If you are new to macros in any scripting language, then this may sound a bit confusing. Let's go through a few examples to show what the above means.

"Macros are used to insert code at compile time"

What this means is that the code you define in a macro will simply be inserted at the location of your `!insertmacro`, as if copy/pasted, when you compile your installer script.

Example 1.1.1

```
!macro Hello
  DetailPrint "Hello world"
```

```
!macroend

Section Test
  !insertmacro Hello
SectionEnd
```

Could be seen as just:

Example 1.1.2

```
Section Test
  DetailPrint "Hello world"
SectionEnd
```

The above is obviously just a simple example with a single line and really wouldn't be a good use of macros. But if you have multiple lines of code that you may have to use over and over again, it may be a good idea to start using a macro for it; it allows you to just make the any changes once in the macro, and it will automatically be changed anywhere you insert it. It also just makes your code look a lot cleaner (a single `!insertmacro` line vs perhaps a dozen lines) which makes it a lot easier to follow and edit.

"depending on defines and using the values of the defines"

That bit is most likely to sound confusing, but gets a little more clear once you read the definition of macros in another section of the NSIS help:

"macro definitions can have one or more parameters defined. The parameters may be accessed the same way a !define would (e.g. ${PARMNAME}) from inside the macro."

If you are familiar with scripting in other languages, this may sound familiar to you, except in the context of functions - but please do not mix this up with Functions as they exist in the NSIS language.. we'll get to those later.

What the above means is that you can have a macro take parameters, or arguments, and use those within the macro:

Example 1.1.3

```
!macro Hello What
  DetailPrint "Hello ${What}"
!macroend

Section Test
  !insertmacro Hello "World"
  !insertmacro Hello "Tree"
  !insertmacro Hello "Flower"
```

```
SectionEnd
```

Could be seen as just:

Example 1.1.4

```
Section Test
  DetailPrint "Hello World"
  DetailPrint "Hello Tree"
  DetailPrint "Hello Flower"
SectionEnd
```

However, you only needed a single macro definition to get these three different results. Let's take a more complex example, straight from the NSIS Help. In NSIS, a Function can only be specified as being for the Installer, or the Uninstaller (prefixed by "un."). The reason for this is that it allows the compiler to make a smaller Uninstaller if it does not need the Installer's functions, and vice-versa. However, sometimes you may have a function that both the Installer and the Uninstaller require. You could then code the Function twice:

Example 1.1.5

```
Function SomeFunc
  ; Lots of code here
FunctionEnd

Function un.SomeFunc
  ; Lots of code here
FunctionEnd
```

However, as it should be apparent, if there's lots of code involved you have two separate areas where you have to make code changes, your code looks less clean, etc.

With the use of macros, this can easily be resolved by writing a macro around the function that simply adds the "un." prefix when requested:

Example 1.1.6

```
!macro SomeFunc un
  Function ${un}SomeFunc
    ; lots of code here
  FunctionEnd
!macroend
!insertmacro SomeFunc ""
!insertmacro SomeFunc "un."
```

The above will then be written out as example 1.1.5, but now you only have a single portion of code to maintain.

For more information on this specific topic, see: Sharing functions between Installer and Uninstaller

Hybrid

Mainly because of the fact that you can't easily pass parameters to Functions, many users adopt a hybrid approach employing both macros -and- functions, and a little bit of a `!define` (a `!define` just allows you to say that e.g. "NSIS" means "Nullsoft Scriptable Install System" without having to write out as much).

Example 2.1

```
!define writeFile "!insertmacro writeFile"

!macro writeFile File String
  Push "${String}"
  Push "${File}"
  Call writeFile
!macroend

Function writeFile
                              ; Stack: <file> <string>
  ClearErrors
  ; Notice we are preserving registers $0, $1 and $2
  Exch $0                     ; Stack: $0 <string>
  Exch                        ; Stack: <string> $0
  Exch $1                     ; Stack: $1 $0
  Push $2                     ; Stack: $2 $1 $0
  ; $0 = file
  ; $1 = string
  FileOpen $2 "$0" "a"
  FileSeek $2 0 END
  FileWrite $2 "$1"
  FileClose $2
  Pop $2                      ; Stack: $1 $0
  Pop $1                      ; Stack: $0
  Pop $0                      ; Stack: -empty-
FunctionEnd

Section Test
  ${writeFile} "$TEMP\a_log_file.txt" "A log entry"
SectionEnd
```

As you can see, this allows you to combine the best of both worlds by reducing code redundancy both in your installation script -and- in the compiled installer and allowing you to 'pass' parameters/arguments to a function by using a macro as an intermediate. And in the end, your code looks even cleaner.

Caveats - or Pros/Cons

Now you might think that macros and functions rather look the same - when should you use which? That all depends on your needs/desires for the most part. However, there are certainly pros/cons to both.

As pointed in the definitions already, there are some immediately obvious advantages over each other:

Macros

- + easier follow (the code simply gets copy/pasted)
- + can easily pass parameters / arguments
- - code gets duplicated in the compiled result

Note that the code duplication is marginal (the files you are installing are typically much larger than the base installer code), and with compression it becomes even less of an issue.

Functions

- + code does not get duplicated in the compiled result
- - cannot easily pass parameters / arguments
- - a bit less easy to follow

(The Hybrid approach pretty much eliminates the parameter/argument passing con of Functions)

However, there are also some less obvious Pros/Cons. For example, Macros are faster to execute than Functions, as there are no opcodes for calling/returning required in the installer. However, this is also marginal and even on older hardware negligible.

Also, code in Functions is not taken into account in the Progress bar, only code in Sections. Depending on what you want to reflect to the user, you may want to move some code into Functions or on the contrary move back code from functions into macros inside Sections.

More intricate Pros/Cons are described in the next sections.

Macros

labels

As a Macro literally gets inserted, or copy/pasted, into your code when compiling you may find yourself running into an issue of duplicating labels:

Example 3.1.1

```
!macro LoopThreeTimes
  StrCpy $0 0
  loop:
    IntOp $0 $0 + 1
    IntCmp $0 3 end
    goto loop
  end:
!macroend

Section Test
  IfFileExists "$TEMP\install.log" end
  !insertmacro LoopThreeTimes
  end:
SectionEnd
```

The above could be read as:

Example 3.1.2

```
Section Test
  IfFileExists "$TEMP\install.log" end
  StrCpy $0 0
  loop:
    IntOp $0 $0 + 1
    IntCmp $0 3 end
    goto loop
  end:
  end:
SectionEnd
```

And the problem should be clear - there are now two labels named "end", which is not allowed. You could rename your labels in your macros to be supposedly unique - in example 3.1.1 you could call them "LoopThreeTimes_loop" and "LoopThreeTimes_end". But what if you use the macro more than once in the same section?

Example 3.1.3

```
Section Test
  IfFileExists "$TEMP\install.log" end
  StrCpy $0 0
  loop:
    IntOp $0 $0 + 1
    IntCmp $0 3 LoopThreeTimes_end
    goto loop
  LoopThreeTimes_end:
```

```
  StrCpy $0 0
  loop:
    IntOp $0 $0 + 1
    IntCmp $0 3 LoopThreeTimes_end
    goto loop
  LoopThreeTimes_end:
  end:
SectionEnd
```

You then again get a duplicate label name. A common way to prevent this is to actually make the label names unique for each time the macro is inserted. You can do this by adding a line number that is first defined to the label;

Example 3.1.3

```
!macro LoopThreeTimes
  !define ID ${__LINE__}
  StrCpy $0 0
  loop_${ID}:
    IntOp $0 $0 + 1
    IntCmp $0 3 end_${ID}
    goto loop_${ID}
  end_${ID}:
  !undef ID
!macroend

Section Test
  IfFileExists "$TEMP\install.log" end
  !insertmacro LoopThreeTimes
  end:
SectionEnd
```

Which would then end up as (presuming the line number the !define was on was 55): Example 3.1.4

```
Section Test
  IfFileExists "$TEMP\install.log" end
  StrCpy $0 0
  loop_55:
    IntOp $0 $0 + 1
    IntCmp $0 3 end_55
    goto loop_55
  end_55:
  end:
SectionEnd
```

You can read more about this in: Tutorial: Using labels in macro's

Although this is a reasonably minor Con, given the easy workaround, it should be clear that Functions do not suffer from any issues with labels at all.

Functions

File command

When you use a Macro, the code gets inserted or copy/pasted if you will into the area of use. This becomes especially important when you use the File command. Consider the following two examples:

Example 3.2.1 - Macro

```
!macro InstallFilesTo Path Filespec
  SetOutPath "${Path}"
  File "${Filespec}"
!macroend

Section Test
  !insertmacro InstallFilesTo "$INSTDIR" "data\*.*"
SectionEnd
```

Example 3.2.2 - Function

```
Function InstallFilesTo
  Exch $0
  Exch
  Exch $1
  SetOutPath "$1"
  #File "$0" ; Does not work!
  Pop $1
  Pop $0
FunctionEnd

Section Test
  Push "data\*.*"
  Push "$INSTDIR"
  Call InstallFilesTo
SectionEnd
```

~~The two perform exactly the same job, namely installing all files in the installer source's "data" sub-folder to the installation folder.~~

The big difference is that with the Macro variant, the size of the files can directly be added to the Section. Which means that your installer will correctly report the amount of free space required on the drive, and the "SectionGetSize" command will work correctly.

With the Function variant, the compiler doesn't know to figure this out, and the required space is not calculated correctly, nor does "SectionGetSize" work correctly.

A workaround for this would be to use "SectionSetSize" where appropriate, but it does mean that you would need to know the size of the files -before- you use "SectionSetSize", which would have to be handled by a separate NSIS script, or some cheating by extracting the file before actually installing, getting the file size, and then moving it or deleting it when installing/not installing. In other words, there is no easy/proper workaround for it.

Conclusion

Although Macros and Functions are very similar beasts, each has its strengths and weaknesses, and certainly things to look out for. In general, however, you can follow these guidelines:

- if you're not using the code more than once and it's just a single line or a few lines, don't bother using either a Macro or a Function.
- if you're not using the code more than once, but it's a good number of lines and is really a procedure that you want to easily be able to maintain or is otherwise just clogging up the view of the surrounding code, consider putting it into a Macro.
- if you're using non-trivial code more than once, always consider putting it into a Macro or Function
- if it's a lot of code (many, many lines), consider using a Function rather than a Macro
- if it's code that gets used only once in the installer -and- only once in the uninstaller, consider using a macro rather than a function, as this saves un.Function hassles
- if you need to use the File command, consider using a Macro rather than a Function - or, if possible, re-think the process to keep the File command outside of the Function.
- if you don't want to fuss around with labels, consider using a Function rather than a Macro - or consider using the work-around which is really quite painless.
- consider setting up a !define for your Macros to make the code cleaner
- consider using the Hybrid approach whenever you decided to go with a Function but need to pass parameters/arguments