DS Programming 101: From Zero to Hero!

Table Of Contents

Preface

Hello and welcome! If you are reading this then it's likely that you're interested in getting to know more about programming for the Nintendo DS! If you are not, then you likely took the wrong turn, but let's not get into that.

Let's also start with establishing one important thing – as the title suggests, this is a "From Zero to Hero" guide. If you are an experienced programmer then it is likely that you will not benefit from it much, if at all. It is going to introduce the very basics to users who have never even seen a compiler before and never coded in their life – stuff that you probably already know and aren't interested in anymore. You are however still welcome as this is my first tutorial and will likely require a certain degree of proof-reading, plus, you may of course have useful suggestions! Keep in mind the target audience though; I'm doing my best not to introduce complicated concepts early on. If you're not an experienced programmer or never programmed at all, this is a great place to start!

I've seen many guides approaching this subject – some were more helpful, some were rather vague, but there is one thing that was common in all of them, and it became apparent to me that something has to be done about it.

The guides I've seen so-far are dedicated to users who are familiar with programming and only require an introduction to the DS environment; none of them are actually "tutorials" from the ground up. Does this mean that a non-experienced user simply cannot program for the DS or should not begin his adventure with programming on this exact platform? No, it does not! In fact, the DS is likely the easiest platform to program for when it comes to consoles – libnds is really not that hard to wrap your mind around and there are numerous libraries out there that facilitate programming for it even further.

You probably want to ask: "If it's so easy, why do you think it requires some sort of an explanation? The libraries are well-documented, do you expect the readers to be dill-wits who can't follow simple examples?" and the answer to that is "No, in fact, I do believe that everybody is capable of programming, however one has to learn and acquire some basic programming habits and have some practice in C to be successful at it" and this is exactly the main goal of this tutorial. Depending on the interest shown by users and my workload at Uni this may or may not be a full-featured guide, however I promise that I will at least try to keep it up-to-date and expand upon it from time to time.

Now that the purpose is established, let's move on to the juicy parts! I hope you will enjoy learning together and in case of any questions or suggestions, do write! Dear readers, keep in mind that the first few tutorials will be an incredibly rapid course in C, applicable to any type of programming, not just for the DS! We won't be compiling much until this material is covered and thoroughly understood! So… Let's get it on!

Chapter 0: Preparing the Environment

Alright then! We've established our goal; it is time to make the first steps towards it! For the duration of this course we will be using the **C language** which is versatile and simple enough to understand. We'll program in the **Visual C++ Express 2008** and we'll be using **NFLib by NightFox and Co.** as our graphics library so that graphics do not impede our progress, but we're miles before getting there. First and foremost, we need to start up our download queue, and we'll need plenty-fancy things!

Firstly, we of course need the **devKit Pro toolchain**, available here:
    http://sourceforge.n...ed%20Installer/
Next, we need to get our hands on **the code editor**:
    http://msdn.microsof...future/bb421473
We will also need **Drunken Coders's Wizard** to add a Nintendo DS application (among others!) as a project type:
    http://pern.drunkenc...udio_wizard.zip
Finally, we'll download **NightFoxLib**:
    http://sourceforge.n...ts/nflib/files/

Once all the files are downloaded, proceed to installing them, starting from devKit Pro (preferably on C:\), next the Visual C++ Studio Express 2008, the Drunken Coders's Template Wizard and finally NightFox Lib (preferably in the devKit Pro directory).

Now we have our programming environment ready! Exciting, isn't it? Well, not quite. Why? Because now we have to move onto the Introductory chapters of this guide – they may be boring and taxing but without them we will not be able to really program anything as everything you will be reading from now on will sound like black magic before this knowledge sinks into you, at least partially. Thus, without further ado, we move onto…

Introductory

Chapter 1: Variables!

To even have a chance at programming something, we need to know what we'll be dealing with. **Introductory Chapter 1** will introduce the first important subject – **Variables**, your main weapons when it comes to programming.

So, what **is** a Variable? Well, in layman's terms, a variable can be just about "anything" – **it can be a number, a character, literally anything,** but to simplify it, we'll say it is a **value**, a **piece of data** that is kept in the console's memory for later use. In most tutorials you would see a link to a Wikipedia article or a reference to a book that will introduce you to the subject – not here. We want to get from Zero to Hero and we want to get there fast, don't we? As far as we're concerned, we could possess all the knowledge about the variables and we still won't be able to use them anyways! I will only pass onto you the knowledge that will actually be relevant to you, so bear with me.

Let's start with describing how to declare a Variable:

```
Type Name = Value;
```

Your typical variable has four identifying features – **a scope, a type, a name and a value**. First and foremost we will tackle **types**, as they determine how we will actually use a given variable.

We can divide our variables into 3 main groups – **Integers, Booleans and Chars**.
Those can be **Signed** or **Unsigned** (except Booleans), plus Integers may **range from 8 to 32 bit** ones (even 64 on PC's, but let's not go ahead of ourselves). Sounds like black magic so-far? Good, cause we're going to explain what all that means, starting from types and their properties:

Signedness is the first property we will discuss. One could write a whole elaborate essay on the differences between signed and unsigned variables. What you need to know is that Signed variables range from negative values throughout 0 and positive values while unsigned ones range from 0 upwards. That's it, really. That's all you need to know!

Next, we'll tackle the lovely **Integers**. As I said, Integers on the DS may consist of **8, 16 or 32 bits**, but **what does that mean**, exactly? Well, it means what **range of values a variable will be able to carry!** For example, an **8-bit integer** will have a length of **255**, meaning it will range from **-128 to 127** when signed and from **0 to 255** when it's unsigned. A **16-bit** one will have a length of **65,535** while a **32-bit one**, a length of **4,294,967,295**.

To use them, we will have to learn how to **declare** them! The symbols used to refer to those integers are u for Unsigned, s for Signed, followed by the length in bits, for example "**u8**" or "**s16**".

Next we have **Booleans**. There's really not much to say about them other then they may hold only two possible values – "**true**" (1) or "**false**" (0). We will mostly be using them for simple switches and there's no point in dedicating any more time to them. The symbol used to declare Booleans is "**bool**".

Finally, we have **chars**… **Chars** will mostly be used to hold **characters or numbers**, but they are much more versatile then that. By "**holding anything**" I really meant that you can use those variable to hold "**anything**". From **text** to **graphics** and **sound**, arrays of chars **will be your tool of choice**. The symbol to declare chars is "**char**".

Now that types have been discussed, we can move to **Scopes**, and we'll discuss only two: **Global** and **Local**.
A **Global Variable** is declared **outside** the main block of the program, thus can be called anywhere within it while a **Local Variable** is declared **within** a function, thus may only be used within that function. I'm sounding vague, huh? Functions? Blocks? **What am I talking about?** Well, let's get right onto that in the **Introductory Chapter 2**, as I don't think the "name" part needs any explanation other than "don't use special signs or spaces when naming a Variable".

Chapter 2: Functions!

So, **what is a Function**, you ask? Well, a Function in layman's terms **is a set of instructions that the program is supposed to carry out**. How does a function look like, how do we declare it you ask? Let's see the declaration for ourselves:

Type Name(TypeOf Argument, TypeOf Argument2){ … };

That… looks incredibly complex, doesn't it? Let's cut it into pieces, shall we?

Where "**Type**" you put either "**void**" if you don't really want the function to return any value for you, just carry out the instructions. If you do want it to return some sort of a value or data, you can specify it using the **previously described Types** – that way, you will be able to assign the value returned to a given variable! Snazzy!

The **Name** can be just about anything, but without spaces and really fancy special signs.

As far as **Arguments** are concerned, **there can be as many as you like** and they can be **any type of a variable you want**.

How do we call our declared function? Well, we simply do this:

NameOfTheFunction(Value1, Value2);

Simple!
Just to practice, let's build our very first simple function, shall we?

CODE

```
s16 Adding(s16 Number1, s16 Number2){
s16 Result=Number1+Number2;
return(Result);
}
```

So, **what does this function do?** Well, it adds the value specified as Number1 to the value specified in Number2 and returns its number as shown by Result!

So, if we do this:

CODE

```
s16 Number3;
Number3=Adding(2, 2);
```

Our function will assign the value "4" to the declared variable Number3. Neato, huh? Of course we're not going to build functions with *that* simple functionality but hey! We're just starting, right?

Before we can actually create any fun functions, we'll have to learn a bit about operators and built-in C functions… **but I'm feeling like you guys are yawning at this point** – so much new-found knowledge and we can't use it anywhere! Annoying, isn't it? Well…
Let's do something else, shall we? Let's **build** our **very first application!** Oh, the **excitement!** First and foremost though, we will analyze the template we're given from libnds so that we know where to put what, hmm? Your minds open? You're fingers ready? This is the part you were all looking forward to, so **let's do this!**

Exercise #1 - Your first program!

Open up Visual Studio C++ Express, start a new project and select the DS as your platform of choice via the wizard – the default settings will do. Once you're done, you will be able to browse the contents of your Solution using the table of contents situated on the left-hand part of the screen. Right now you only have the Source file called **"template.c"**. Let's open it! You will likely see this as the result:

CODE

```
 /*---------------------------------------------------------------------------------


        Basic template code for starting a DS app

---------------------------------------------------------------------------------*/
#include <nds.h>
#include <stdio.h>
//---------------------------------------------------------------------------------
int main(void) {
//-------------------------------------------------------------------------
        consoleDemoInit();
        iprintf("Hello World!");
        while(1) {
                swiWaitForVBlank();
        }

}
```

Wow wow wow, that's alot and sooo fast! Let's look through this quickly, shall we?

Every program is essentially divided into three main sections: the **Includes**, the **Declarations and Defines** and finally the **main();** Function. We will discuss them all briefly. I will also explain the three functions that appear in this code so that they're **crystal clear**.

#include is a directive referring to any files that may be included in the project. It's vital to know that #include files will *always* be in our RAM – their content can be accessed anywhere throughout our program! This will turn out to be very useful when organizing your work and resources later-on, so remember it! The two files included are nds.h, which is the libnds library and stdio.h, which is the Standard Input/Output library of C and C++. The Angle-Brackets <> indicate that the compiler will search for the file using it's specified INCLUDE path as well as the parent directory of your program, were we using simple quotation marks "", the compiler would attempt to find it only in the parent directory, but more on that later.

Normally after the #include we would start putting our **Global declarations**, every variable declared there will be treated as Global. Keep that in mind!
This basic template has no variables, but let's add one, just for the heck of it! Scroll up and put your declaration right before the main(); function. Let's say…

char MyName**[25]**="Put your name here, silly!";

Can you tell me what kind of a variable that is? **Yep, you are absolutely right!** Great attention span! It's indeed a char, its name is "**MyName**" and its value is… well, whatever your name is! What's with the weird number in the brackets, you ask? Well, one char can only hold one character – with this number, our variable is now an **Array**, otherwise known as **String**, of the **length of 25 characters** – that way, we can put more characters in, but more on that in the later chapters about Arrays. In any case, **good job!** Remember about the "**;**", without putting it at the end, the compiler will not know when the line of input ends and will return an error, so **declare carefuly!** Let's move on.

Lines starting with "**//**" or enclosed between "**/***" "***/**" are called **Comments** – the compiler ignores them, but they are helpful to mark important parts of your code.

Finally we move on to the **int main();** the most interesting part we'll tackle today.
main(); is effectively the main function of the program – whatever is supposed to happen is placed in it. It is divided into two main parts – the part before **while(1)** and after it. We'll talk about the while loop later, for now, all we need to know is that whatever is before while happens once, whatever comes within its brackets happens once each frame (the DS works at circa**60 frames per second**, so the maths here are clear).

Before while(1) we have two functions, let's have a closer look to know what they are:

consoleDemoInit(); is a function built-in libnds, it will launch a DOS-like command line to which you will be able to output text. Nothing more, nothing less, really.

iprintf("Hello World!"); is a function used for printing text. It prints it directly into the console, the text printed is placed between the quotation marks. Simple!

Let's head to while, shall we?

It contains only one function - **swiWaitForVBlank();** which, as the name implies, wait for a VBlank, meaning the end of a frame. You will have to put this function at the end of your main, always!
So, from what we've read so-far, this program should:

1. Initialize the console.
2. Print Hello World! On the screen.

Let's see if it does that, shall we? Select "**Build**" from the context menu at the top of the screen, then hit "**Build Solution**", alternatively just hit "**F7**". This will begin the makefile operations and compile your very first .nds file. **Excited? You should be!**

Once Building Operations are finished, you will be able to read where the file was saved at the bottom part of the screen. **No surprise there** – it's in the folder you specified when creating the Project.

Open that folder – the .nds file should be there. Provided you have an .nds emulator, you will be able to launch it, and… **magic!** It does what it's supposed to!
I can hear you nagging "but what about my Variable?!? I want it to do something that wasn't in the template, you're a lame teacher!" and that **breaks my heart.** Would I desert you? Would I? **Never!**

Let's **spice up the code** a little bit, hmm? Let's make this program specifically about ***you***.
Scroll back up where you declared your first variable and declare another one, like this:

```
char Text[100];
```

We already know it's a string of characters with a specified length, but why No value? Well, it means that the string is **empty** – we can put data into it later-on though. This is called a "**buffer**" – a place dedicated in RAM for us to input data into.

Let's move onto the function though, shall we? Before the **iprintf()**, we'll add another fun function that is commonly used with strings:

```
sprintf(Text, "Hello World! My name is %s!", MyName);
```

Okay, **confusing**, I know. Explaining now. **sprintf();** is a function used for formatting strings – you select a **destination buffer**, then you write the **format** within the brackets, finally you write**which variables should be used to format it**. In this case, we are inserting the string MyName using "%s". The final result should be "Hello World! My name is Whatever you specified as your name". That's **neat**, isn't it?

Now, let's modify the iprintf to use our text. All you need to do is this:

iprintf(Text);

Now, rather then Hello World!, the text printed will be whatever has been put in the **buffer Text** (hence the lack of brackets). Now, what you should have is this:

CODE

```
#include <nds.h>
#include <stdio.h>
char Text[100];
char MyName[25]="Foxi4";
int main(void) {

        consoleDemoInit();
        sprintf(Text, "Hello World! My name is %s!", MyName);
        iprintf(Text);
        while(1) {
                swiWaitForVBlank();
        }
}
```

Ready? Steady? **BUILD!** So… does it work? Well, why shouldn't it?

You've just built your very first program for the DS, **congratulations!** Admittedly it's simple, but with some dedication, soon enough **you'll be a pro at this!** One thing is for sure, **you're no longer a Zero**, you moved onto… **0,1**… So **there's still a road ahead of you!**

Chapter 3: Operators in C

So far, we've learned the basic structure of an application in C, the types of Variables and looked at some functions - I hope everything so-far sunk in because we're about to look at **if statements** and **switches** with various **conditions!** Before we get there though, we need to know how to build those **conditions**, and for that, we'll need to have a look at **Operators**.

Without further ado, let's get to it, starting from explaining what exactly they are.

Operators are very simple elements which tell the program **how to combine, compare or modify Values**.
By putting them in-between of two **Operands** we create **Expressions**. We can also use some of them **connect Expressions together**, designing more advanced **Logic**.

Let's use an example that's known from our daily life and math - an **expression** is for example "3+4" where "+" is the **Operator**. **Operators** are divided into **5 groups**, each of them with different uses.

Those groups are **Arithmetic, Comparison, Logical, Assignment and the Ternary Operator**. First, we'll have a look at **Arithmetic Operators** as we've already used a few. They are used exactly the same as you'd use them in everyday math tasks, so their function should not pose any problems to anyone. Those Operators are:

ARITMETHIC OPERATORS

+ addition
- substraction (- when placed before numbers negates them)
***** multiplication
/ division
(and **)** define precedence (to define in what order the calculations happen)

++ incrementation (adding 1)
-- decrementation (substracting 1)
>> bit shift right
<< bit shift left
% modulus (calculating the remainder of dividing two numbers)

Example of use: *AVariable=3***+***3;*
(Assign the result from adding 3 to 3 to AVariable)

As of now, only the first 5 (well, maybe 7 😋) are of our interest, the rest will be covered in more detail in subsequent chapters if they pop up, for now you just need to know they exist.

Next up are operators which we also used, called **Assignment Operators**. We use them to **assign Values** to a given **Variable**. Those operators are:

ASSIGNMENT OPERATORS

= Simple assignment. The Value of the right operand is assigned to the Variable on the left.
***=** Multiplication assignment. The Value of the left operand is multiplied by the Value of the right operand and the result is assigned to the left one.
/= Division assignment. The Value of the left operand is divided by the Value of the right operand and the result is assigned to the left one.
%= Remainder assignment. The remainder of dividing the Value of the left operand by the right operand is assigned to the left operand.
+= Addition assignment. The Values of operands are added and the result is assigned to the left one.
–= Subtraction assignment. The Value of the right operand is substracted from the Value of the left operand and the result is assigned to the left one.

<<= Left-shift assignment. The Value of the left operand is shifted left by the ammount of bits specified by the right operand and the result is assigned to the left one.
>>= Right-shift assignment. The Value of the left operand is shifted right by the ammount of bits specified by the right operand and the result is assigned to the left one.
&= Bitwise-AND assignment. Obtains the bitwise AND of the left and right operands and assigns the result in the left operand.

^= Bitwise-exclusive-OR assignment. Obtains the bitwise exclusive OR of the left and right operands and assigns the result in the left operand.
|= Bitwise-inclusive-OR assignment Obtains the bitwise inclusive OR of the left and right operands and assigns the result in the left operand.

Example of use: *AVariable***=***AnotherVariableOrAValue;*
(Assign the value of AnotherVariableOrValue to AVariable)

Once again, I've divided them into two groups - the ones you should be concerned with firstly and the ones that we'll be using much, much later.

Now we can move on to what interests us the most today, which would be **Comparison Operators.** These will be the basis of building conditions and you need to know them by heart. Their purpose is to compare two **Values** and return the result of this comparison. Those operators are:

COMPARISON OPERATORS

== Checks if the Value on the left is equal to the Value on the right.
!= Checks if the Value is not equal to the Value on the right.
< Checks if the Value on the left is smaller then the Value on the right.
> Checks if the Value on the left if greater then the Value on the right.
<= Checks if the Value on the left is smaller or equal to the Value on the right.
>= Check if the Value on the left is greater or equal to the Value on the right.

Example of use: *if(Variable***<***3) Variable++;*
(if the Variable's value is less then three, increment it)

ALL **Comparison Operators** are essential; there is literally nothing we could leave for later.
Finally we can move on to another useful kind of **Operators** which we will frequently use to join together different expressions. Those operators are called **Logical Operators** and are relatively simple to remember since there are only three of them.

LOGICAL OPERATORS

! Is used to negate a given expression, this operator is called NOT.
&& If both parts of the expression are true, the whole expression is true. If either part if false, the expression will be false. This operator is called AND.
|| If either part of the expression is true, the whole expression is true. This operator is called OR.

Example of use: *if(AVariable==3* **&&** *AnotherVariable==3) AVariable++;*
(if both variables are equal 3, increment the first one)

That'd be all we need to know now - the **Ternary Operator** will be introduced together with if statements. That's all for this chapter and tune in next time to learn how to build Conditions with our new-found knowledge, alongside some fun examples!

Chapter 4: Conditions

Now that we know **Operators**, we are pretty much set to start learning about creating various **Conditions in C**. What is a **Condition**? Well, it is a **Statement** that specifies **how our program is going to react when a given event happens**. We can divide those statements into two basic groups, **if Statements** and **switches**. Firstly we'll have a look at both, then we're going to compare them as to know exactly when to use them efficiently.

We've actually seen a few **if statements** already in the previous Chapter, I'm sure you've noticed! These were however very short - they were the shortest possible form of that kind:

```
if (Condition) Result;
```

With that kind of a statement we can only specify **one Result** and **one Condition**, so naturally this is a **really limited** statement. To fully utilize **if Statements** we need to be familiar with their entire structure, which is as follows:

CODE

```
if (Condition){
   Statements;
  }
else if(Condition2){
   Statements;
  }
else{
   Statements;
  }
```

We can see that this is a **much more flexible** implementation of an **if Statement**, we can not only specify as many **Conditions** and **Results** as we want using **else if**'s, we can also set a **Result**that will occour **if neither of the Conditions are met!** It is divided into three parts, our first **if**, a number of following **else if's** and finishes with **else** - the latter two parts are **entirely optional**and you can ommit either of them if you don't need them.

To show how this works in a more practical way, let's have a look at this example:

CODE

```
if (Age>=18){
   iprintf("You're an adult!");
  }
else if(Age<18 && Age>=0){
   iprintf("You're a minor!");
  }
else{
   iprintf("Very funny. You were supposed to input your age!");
  }
```

Here, if the **Variable** Age's **Value** is **greater or equal** to 18, the program will inform the user that he is an adult by printing a message on the screen. If the Value is **less then** 18, but more then 0, the program will state that the user is a minor. If the Variable has a value that is **incorrect**, such as less then 0, the program snarkly remarks that the user inputted an incorrect value.

Of course we could just do this:

CODE

```
if (Age>=18){
   iprintf("You're an adult!");
  }
else{
   iprintf("You're a minor!");
  }
```

But this presents a **degree of ambiguity** - if the user would input a value that is less then 0, the program would **still** refer to him as a minor **despite** the fact that the value is **incorrect**. This could be prevented by doing this:

CODE

```
if (Age>=18){
   iprintf("You're an adult!");
  }
else if(Age<18 && Age>0){
   iprintf("You're a minor!");
  }
```

However again, the program would be confused if the inputted value was of **incorrect type**, for example a character rather then a digit and this would result in a **glitch** - the character would be transcribed into a numerical value.

Why am I saying all this? You're going to say *"we're not brain-dead, this is obvious!"*. Well, let me tell you, **it's not.** I'm doing this to teach you to **avoid being ambiguous at all costs,** this will save you time when **debugging** your applications.

Keeping these issues in mind will help you **avoid glitches** in your program and prevent it from acting in a manner that you would not expect. **Computers are not intelligent,** they will not do your work for you.
You have to imagine **you're giving orders to a complete simpleton** who requires really specific instructions as to **not make a mistake.**

Simplify your code **when it is possible, by all means,** but **where there is room for error,** specify your intentions **in detail.**

To conclude the section about **if Statements**, we'll have a look at the **Ternary Operator** as promised last time. Sometimes our conditions **can be thoroughly simplified** and we can ommit the use of **if Statements** or **switches** altogether by using the **Ternary Operator "?".**

Let's consider this snippet of code:

CODE

```
if (Variable==10){
        Result = 1;
}
else{
        Result = 0;
}
```

We created an **if Statement** which checks whether the **Variable** is equal to 10. If it is, the program will assign the value 1 (true) to **Result**, otherwise it will assign 0 (false). We can simplify this with a simple **Ternary Operation**, like this:

CODE

```
Result = Variable==10 ? 1 : 0;
```

This will automatically assign the values to **Result** depending on whether the **Condition** will turn out to be **true** or **false!** Much quicker, **isn't it?**

Now that we are accustomed with **if Statements,** we can learn about **switches**, which are very similar in function however may come in handy simply because they are more **see-through** and**easier to debug.** Let's have a look, shall we?

CODE

```
switch(Variable){
  case 1:
        Statements;
        break;
  case 2:
        Statements;
        break;
  default:
        Statements;
        break;
}
```

In this example **switch**, we have two **cases** and the **default case**.
If the **Value** of **Variable** is **equal to 1**, it performs the **Statements** specified in **case 1**, if it is **equal to 2**, it performs the **Statements** specified in **case 2,** if it is **neither,** it performs the **Statements** specified in the **default case,** which is **optional.** Each case is separated by **break;** as to **conclude the case,** however it is **optional** if you want to receive the same result in **several cases,** like here:

CODE

```
switch(Variable){
  case 0:
        iprintf("Variable is equal to 0");
                break;
  case 1:
  case 2:
        iprintf("Variable is equal to either 1 or 2");
        break;
  default:
        iprintf("Variable is neither 0 nor 1 or 2");
        break;
}
```

By not adding the **break;** after **case 0,** we specified **the same Statement** for both **case 1** and **case 2,** thus saving time and **simplifying** the **switch**.

The **default;** case again is optional, **however** I'm reminding you about possible **ambiguity of code.** This is a **disadvantage** of using **switches** - if the **Variable will not fall in either case specified** and there will be **no default case present,** the program will simply **ignore** the input altogether! Another disadvantage of **switches** is that the **Variable** used must be a **simple Variable** - either an **Integer** or a **char**, thus **we cannot use it with Strings or Structures**.

This concludes **Introductory Chapter 4**, in the next and final introductory chapter we will have a look at **Loops**, specifically **for Loops** and **while Loops.** The rest of C-related material will be discussed as we progress **if needs-be.** Thank you for reading and **see you next time!**

Chapter 5: Loops

Today we'll be having a look at **Loops**, to be more specific, the **for()** and **while()**. In programming, we use **Loops** to create **cyclical action** - we put **functions** we want to use **repeatedly** in them to save time. As always, we'll have a look at both and compare their features to know when to use each of them. Let's start with the **for()** loop and its syntax, shall we?

for(Starting Expression; Testing Expression; Count Expression){...}

Starting? Testing? Counting? What's all that? Well, let's have a look at a more practical example:

CODE

```
u8 i=0;
for(i=0;i<5;i++){
iprintf("This is a line of text.\n");
}
```

As we can see, we created a **Variable** to be used in the **Loop** called **i**, this **Variable** is **incremented** once each cycle as long as the **Testing expression** is **true**, which here means as long as it is less than 5. Until it reaches this state though, every **Statement** between the **{ }** brackets **will be executed once each cycle** aswell! Thus, the end effect is:

CODE

```
This is a line of text.
This is a line of text.
This is a line of text.
This is a line of text.
This is a line of text.
```

You're going to ask "where's the weird "\n" you've ended the String with, huh? Well, this is the **Newline Sign**! Using it you can make sure that whatever new string you'll be inputting into the console will be printed **in the next line**! Let's compare this **for()** to a loop we already know, the **while() Loop**.

while(Condition){...}

Looks a whole lot more simple, huh? Well, it also works **slightly differently**. As you can see, there is no place to put in the **Start Condition**, nor do we have a place where we could **add or substract** from our controlling **Variable**... How do we use it then?

The while **Loop** will execute the actions nested between its **{ }** brackets as long as the **Condition specified is true**. For example...

CODE

```
u8 i=0;
while(i!=5){
iprintf("This is a line of text.\n");
i++;
}
```

The idea here is clear - we have our **Variable i** which we use as a controller. The **Loop** prints text into the console with each cycle and then increments the **Variable i**. After 5 cycles, **i** reaches 5 and the **Looped functions** are no longer executed... Yes, **you guessed it!** The result is:

CODE

```
This is a line of text.
This is a line of text.
This is a line of text.
This is a line of text.
This is a line of text.
```

As you've seen, we can in fact utilize both **Loops** to do the exact same thing, but you will also notice that the **for() Loop** is very much independent - it sets the value for **i** itself and it also sets how this value will change each cycle.

With **while()**, you have to plot the activation and deactivation of the **Loop** yourself.

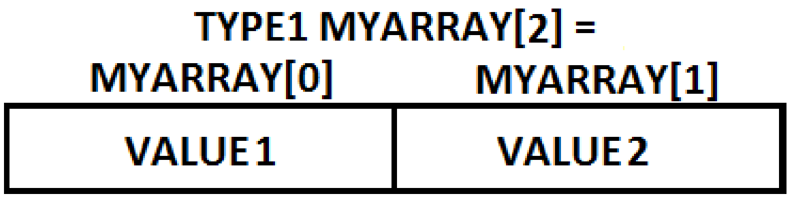
Both cases have their advantages and disadvantages so all I'm really going to say is that we will use **for() Loops** whenever we want a certain function to be repeated a specified number of times and leave the **while() Loop** for functions that only need to be executed repeatedly when a given event occurs, for example when constructing A.I's, but more on that in the Chapter about game logic later on.

I hope you enjoyed this chapter, even if it's a little bit short. There really isn't much to say about **Loop**, so bare with me. I'd also like to inform that there's been a slight change in the schedule - we'll jam a chapter about **Arrays** and **Structures** before we proceed to strictly DS-specific programming as I don't want to run into unintroduced concepts while we're at it. I'll do my best to introduce both briefly and clearly. Stay tuned and thanks again for reading, see you next time!

Chapter 6: Containers of Variables - Structures and Arrays

Structures and **Arrays** are nifty **containers** which will greatly improve our overall work. Rather then working on one **Variable** at a time, we can construct **Arrays** and **Structures** to work on **multiple Variables** at the same time. We've already seen a few **Arrays**, but we never really discussed how they work so we'll start with them.

In essence, an **Array** is a **uniform set of numerous Variables of the same Type** placed next to each other **that can be referred to using the same name.**



We already know how to **declare** an **Array** from the previous chapters:

```
Type ArrayName[NumberOfUnits];
```

But this is only one of possibilities. An Array declared in this fashion will have **no values assigned**; just **empty slots** ready to receive them. You can just as well do this:

```
Type ArrayName[]={1,2,3};
```

Here, we are not assigning a **"total size"** at all - the compiler deduces on its own that initially the Array has 4 slots - for the 1, 2, 3 and the NULL - there are no empty slots.

What's interesting is that **you don't have to use all the values in an Array immediately.** You may assign values to it later until you reach its maximum **Size**. For example, you may declare your **Array** as one with 255 slots while in reality you will be using less then that and add values further as the program progresses:

```
Type ArrayName[255] = {1,2,3};
```

Here, we **assign** values - 1, 2 and 3 to corresponding slots 0, 1 and 2, however, the number of slots is higher than that, thus **the rest is left empty** and a **NULL** is appended to them. Keep in mind though that going past the specified **Size** of the Array will result in assigning **out-of-bounds values!** You **don't want that to happen!**

All that said... why are **Arrays** useful? After all, we could just as well declare the exact number of **Variables** we need, surely declaring them in an **Array** has some advantages? Well, **sometimes you may hold numerous values that refer to pretty much the same thing** - having the capability to call them by the same **Name** is pretty useful. Earlier we saw **Arrays** of char's - the **Array** was a **collection of signs** that were later called as a whole String. Another use for Arrays is that their Values can easily be changed using Looping, like this:

CODE

```
for(i=0; i=5; i++){
MyArray[i]++;
}
```

With only a few lines of code **we've incremented 6 Variables** - **MyArray[0]** up to **MyArray[5]**. Normally it would take us **6 lines of code**, here we're just looping through an **Array** and calling its elements using the **Variable i**.

This shows us how to **call specific units** within the **Array** - all you have to do to call up a given **element** is putting **the appropriate number** corresponding to the unit into the **brackets []**. As you can see, an **Array** can be **called as a whole** as previously shown, but at the same time **you are not losing the capability to call its specific elements** like you would normally call Variables, which is a big plus.

Arrays have another interesting feature. What we declared was a **single-dimensional Array** - a single **row of Variables**. **We are not limited** to this though - you can declare multi-dimensional Arrays easily:

MyArray[**Rows**][**Columns**];

This is for example a **two-dimensional Array**. Try to imagine how the **Variables** are situated in it. Why of course, on a **two-dimensional grid**. This kind of an **Array** could be used for example to track two values referring to the same object - for example the height and width of a sprite or its position on the screen.

I won't dwell into the details here - **the possibilities in design are endless**. What I will mention is the **limitation of Arrays**.

As I previously said, **every element of an Array is of the exact same type**. This can be quite a hindrance as even though **Arrays** are really versatile for holding data, all the data within them has to follow its strict **Type rules**. What can we do when we require a specific variable that will hold data of two or more types? Why, **we create a Structure!**

There are numerous ways to create a **Structure**, but I will only cover my favourite and most see-through method - **creating a new type**. It is only logical that once no available Type can suit your needs, you create a new one, **isn't it?** How do we **declare a new type?** Quite easily, really:

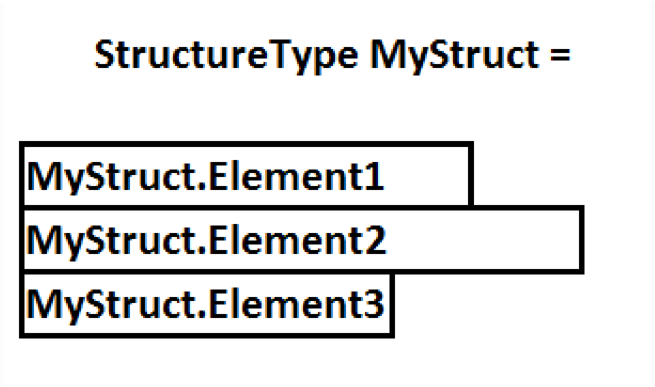
CODE

```
typedef struct{
type1 Element1;
type2 Element2;
type3 Element3;
} NameOfType;
```

As you can see, **the syntax is quite clear** - we create a **definition of a type**, and it is going to be **a structure**. The structure type in this template has three elements and its **name** is**NameOfType**... but this alone is just a type, how do we declare this structure? Easier than you think:

NameOfType **MyStructure**;

Yep - you use your newly-defined **Type** as you would with any other. **How do you call the elements?** Also rather easily - you call them using the **name** of the new **Structure** and the **name** of its **element**, here it would be for example:



MyStruct.Element1, **MyStruct.Element2** or **MyStruct.Element3**

By using a **Structure**, you free yourself from the **limitation of the Array** - you can now use any **type** for any **element** freely. Unfortunately, you also **lose** some of the privileges.

You **cannot** for example freely **cycle through elements** with a Loop as shown earlier as they are not numbered - they have **separate names**.

We'll end our adventure with C here for now... why?
Because **what's coming up is libnds Input!** I honestly believe that the information we've discussed allows us to jump into DS Programming head first without worrying of getting lost in it. The C introduction will still be expanded as we tackle new subjects, but our knowledge is sufficient for now. Can't wait? Me neither! Stay tuned and remember - do comment! **Foxi over and out!**

Practical use of libnds

Chapter 1: Input – Keys and Stylus

With the basic introduction of C behind us, I believe we're **ready** for what you've all been waiting for – the basics of **libnds**.

Until now, even with everything we've learned so-far, we were **unable** to utilize **any** user input whatsoever, as we didn't know **how to use the Keys** or the **Touchscreen** – these are platform-specific after all. Today we're going to learn **how to detect the state of Keys and the Touchscreen and how to use it.**

Let's start with **Keys**. First and foremost, to use **Key Input**, we need to **scan** for their use. To do that, we place the function:

```
scanKeys();
```

Within the **main while(); loop** of the program. This will **initialize the Key scanner** and thus allow us to use them as a method of input.

Their use is divided into two portions – **Key Detection** and **State Detection**. These are of equal importance – **one cannot work without the other**, so keep that in mind!

Keys are neatly put into an easy-to-remember structure of **Enumerators**. We're not going to tackle what exactly Enumerators are; all you need to know that **each Key corresponds to a value**, the **Lid** and **Touchscreen taps** are **also** treated as a **Key**. This is the list of all Keys you can use:

CODE

```
  KEY_A = BIT(0)
  KEY_B = BIT(1)
  KEY_SELECT = BIT(2)
  KEY_START = BIT(3)
  KEY_RIGHT = BIT(4)
  KEY_LEFT = BIT(5)
  KEY_UP = BIT(6)
  KEY_DOWN = BIT(7)
  KEY_R = BIT(8)
  KEY_L = BIT(9)
  KEY_X = BIT(10)
  KEY_Y = BIT(11)
  KEY_TOUCH = BIT(12) (Refers to touchscreen tapping)
  KEY_LID = BIT(13)
```

As you can see, each **Key** has its respective **Name**. Using those we can create **Statements** that will depend on their values, but not just them alone. We will also require their **State**, and for that we have a set of useful functions:

CODE

```
keysDown();
keysHeld();
keysUp();
keysCurrent();
```

Down, **Held** and **Up** are relatively self-explanatory – Down is the State occurring **directly after a button is pressed**, Held is a State that occurs **when the key is pressed and kept down** and Up is a State that occurs **when the key is released**. All three return different values when different keys enter the specified state. **Current** is an interesting State, **as rather then Returning a value to a specific State immediately, it first detects which state it is**.

Now that we know that, we're able to compose our first condition based on **Key input!**

```
if(KEY_A & keysHeld()){
Statements;
}
```

This if statement is quite clear – **if** the **A button** is **held down**, the Statements within the **{ }** will be executed. The **Statements** will be executed until the **State** of the **Key** changes, in this case, when it is **released**. You can do the exact same thing with any combination of button + state. To make it easier and more sensible, it's worth to declare some special Variables before you begin your program

CODE

```
u16 Pressed;
u16 Held;
u16 Released;
```

...and update them at the beginning of every while(1); cycle.

CODE

```
Pressed = keysDown();
Held = keysHeld();
Released = keysUp();
```

This way we save valuable calculation power by checking the **States** of buttons once each frame at the beginning of the program rather then with each button press.

```
if(KEY_A & Held){
Statements;
}
```

Now that we've covered **Keys** we can safely progress to the **Touchscreen** which is also quite simple to use. We'll start by declaring a **Variable** to hold all the data recovered from the screen:

```
touchPosition TouchStructure;
```

We are yet unfamiliar with the type touchPosition - it's a type declared within libnds and platform-specific. It's a Structure type and holds the data on numerous stylus readings:

CODE

```
u16 px (Pixel X value)
u16 py (Pixel Y value)
u16 rawx (Raw X value)
u16 rawy (Raw Y value)
u16 z1 (Raw cross panel resistance)
u16 z2 (Raw cross panel resistance)
```

rawx and **rawy** are exactly what their names imply - raw values read from the screen. What makes them raw? Well, they're not exactly user-friendly as they are not translated to **pixel positions** on the screen, they're large and require further calculations to be used in pointing and clicking, however they are read much quicker then other values, thus are useful at for example implementing **swiping and dragging** with the stylus. **px** and **py** values are calculated from raw values and refer to specific pixels on the screen. The screen is **192 pixels high** and**256 pixels wide**, with the top left-hand corner marked as **pixel 0,0** - knowing this alone will help you use the **px** and **py** values successfully. As for **z1** and **z2** values, they refer to the panel resistance and are useful (from what I know) at reading the pressure, however it's not something you'd normally use and thus we won't talk much about them.

You can use any name you feel comfortable with in this declaration rather then "**TouchStructure**", it doesn't in any way influence the readouts... which we're not getting yet anyways.

To receive readouts of the stylus position, we need to initialize a screen scan function at the beginning of our **main while() loop** as we did with the keys. In case of the touchscreen, we use:

```
touchRead(&TouchStructure);
```

Sigh, I know what you're going to ask, oh-ever-so-inquisitive reader. **"What's that & doing there? That shouldn't be there, right?"**. Well, yeah, it should. This "&" refers to a **Pointer**, you don't know what those are yet and for now you don't need to. What I will tell you though is that to save space and calculation power, rather then using the whole structure inside this function, we just **Point** at the location of the structure in memory with a **Pointer**.

After using this function, all the readouts from the touchscreen will be immediately sent to the **TouchVariable** structure, out of which we'll be able to draw elements as we normally would with a **Structure**, as we already know its elements. For example, the position of the Stylus Horizontal-wise and pixel-wise will be kept under **TouchVariable.px**.

This concludes this Chapter, I hope it was a pleasant read. I'll do my best to include a nice exercise concerning **Key** and **Touchscreen** use soon, for now, experiment on your own and if you have any questions or if you'd like to boast a bit with your newly-written code, go ahead and post! It'll **only get better** from now on!

Foxi over and out!