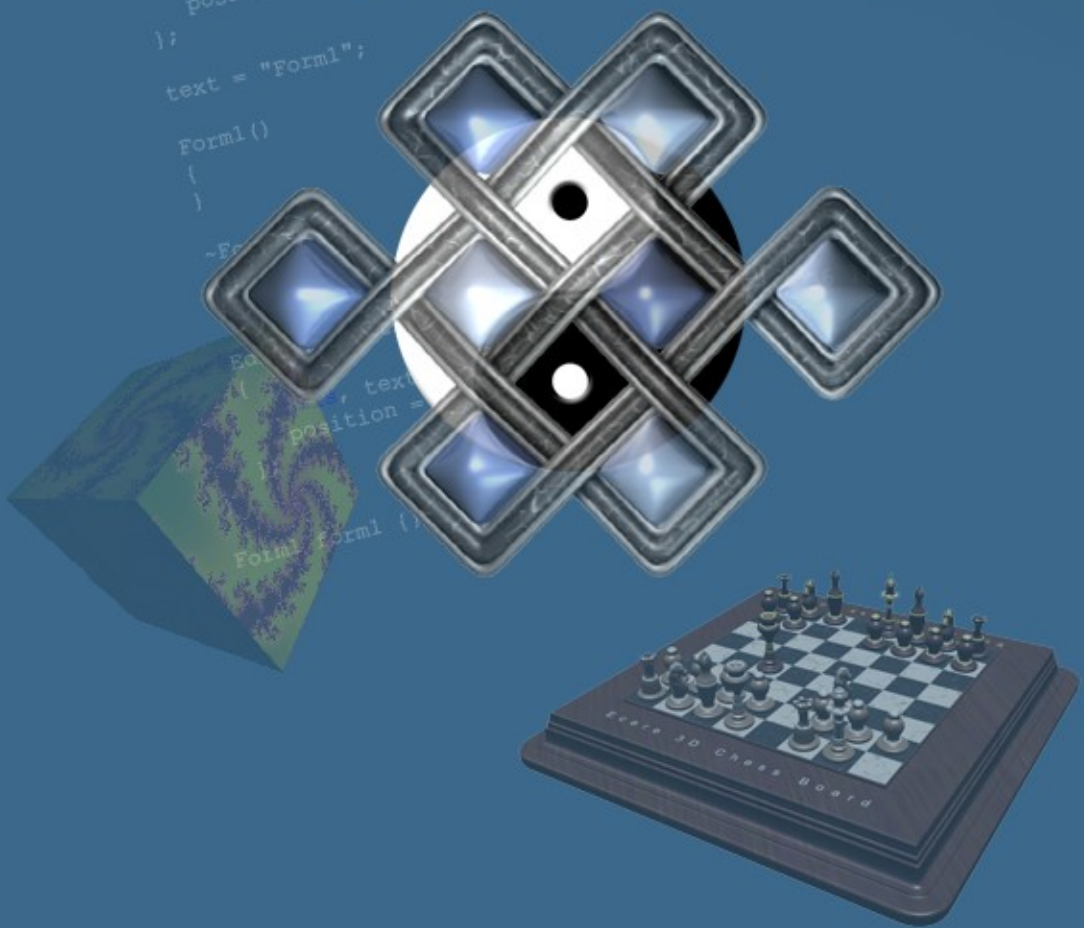
# The Ecere Tao of Programming



Introduction

Installing the Ecere Software Development Kit

Section 1 - Programming in eC
- A very first program
- Variables and data types
- Arithmetic, relational, logical and bitwise operators
- Flow control
- Structures
- Enumerations
- Arrays, pointers and memory

Section 2 - Object-Oriented Programming with eC
- Classes, methods and instances
- Inheritance
- Polymorphism and virtual methods
- Properties
- Encapsulation and access control
- Importing and working with multiple modules
- Name Spaces
- Units and conversion properties
- Bit collection classes

Section 3 - Building Graphical User Interfaces
- Using the form designer and property sheet
- Using static controls such as labels and pictures
- Handling events, push buttons, check boxes and option boxes
- Inputting data through edit boxes
- Displaying message boxes

Section 4 - Drawing Graphics
- Drawing on the Surface
- Using Font and Bitmap resources
- Loading and Manipulating Images

Section 5 - Using the debugger
- Running, breaking, stepping and resuming
- Inspecting the value of variables and expressions

Section 6 - Manipulating Text Strings
- C strings
- Constant strings, dynamically versus statically allocated strings
- Converting between numerals and text
- Browsing through the strings
- Operations with multiple strings
- Working with Unicode to support international languages

Section 7 - Accessing and Manipulating Files
- Opening files
- Reading from and Writing to files
- Manipulating File Names
- Directories
- Temporary Files
- Using archives and project resources
- Listing files and directories
- Spawning other processes
- Monitoring opened files

Section 8 - Network Programming
- Sockets Programming: working with TCP/IP and UDP protocols
- Using Secured Sockets Layers
- Distributed Objects
- Working with XML-based protocols

Section 9 - More advanced GUI programming
- Using data controls such as list boxes and drop boxes
- Presenting common dialogs such as file selection dialogs
- Working with hot keys and labeling controls
- Handling keyboard events
- Taking advantage of anchored positioning
- Building your own controls
- Manipulating the windows
- Interacting with the clipboard
- Working with timers
- Dealing with the time and date

Section 10 - Data Structures
- Linked Lists
- Binary Trees

Section 11 - 3D Graphics Programming
- Basics of 3D Graphics
- Cameras
- Objects
- Materials
- Meshes
- Matrices and Quaternions

Section 12 - Multi threading
- Spawning new threads
- Mutexes
- Using semaphores
- Communicating between threads: a sample event queue

Section 13 - Putting it all together: an in depth study of writing a 3D chess game
- The game of chess
- Building an outer shell interface
- Representing and interfacing with a 2D chess board
- Playing across the network
- Displaying a 3D chess board
- Interfacing in 3 dimensions
- The gift of intelligence

Section 14 - Video Game Type Interaction
- Relative mouse movement
- Interacting with Joysticks
- Working in full screen

Section 15 - Writing database applications
- Designing Tables
- Storing and retrieving data
- Indexing and searching for data
- The cross roads of databases and objects: active records

Section 16 - Custom data types
- Defining custom data types
- Converting to and from text strings
- Serialization
- Comparison and Sorting
- Specifying how to display data in controls
- Using the data box control
- Building custom data types editors

Section 17 - Building custom interface skins

Section 18 - Advanced eC programming
- Reference counting
- Run time class information and generic type parameters
- Observer pattern
- Subclasses, class data and properties
- Loading modules dynamically
- Retrieving and calling methods from class objects
- Building class designers to integrate with the IDE
- Debugging using MemoryGuard

## *Introduction*

Computer programming is nothing less than a form of art. It is an unlimited vehicle for creativity, and also a great inquisitive tool for analyzing our various conceptions of the world. Therefore there are diverse good reasons one might decide to learn programming, each of them probably just as good as the others. Whether you decide to learn it as a hobby or for professional purposes, it is a great skill to acquire which is very likely to prove itself useful in this high tech age. And yes, it occasionally tends to be much fun.

The kind of programming we will study is the classic type. These days many people often answer HTML when asked what language they program in. Hyper Text Markup Language - the core web language used to format the contents of web pages. Yes, some web pages include JavaScript code to perform more advanced user interaction, or are backed by server side programming in PHP, for example. But HTML or XML hardly qualify as "programming" languages (the M stands for "Markup"), and the latter are more oriented towards this whole web back end type of programming. We will focus on building software applications which can stand and run by themselves.

This book will adopt a practical, goal driven approach to learning to program. In each chapter, a clear idea of what is to be learned will be stated, and by the end of it you should have gotten hands on experience and feel confident you have mastered the subject matter.

We will focus strictly on software development using the Ecere Software Development Kit. We will also be writing code in a single programming language, eC. However, if learned correctly, the concepts you will master, the knowledge you will gain, and the way you think in order to solve a problem through programming should be independent from and easily adaptable to any other development tool or programming language you will decide to use in the future.

The Ecere SDK offers a well-rounded suite of tools, which consist of an Integrated Development Environment, a set of compiling tools for the eC language, as well as a cross-platform runtime library featuring among other things a GUI toolkit, a networking library and a 3D engine. It is also backed by the GNU GCC compiling tools suite, which can also be used to program in the C and C++ languages.

This version of the Ecere SDK included with this book is cross platform software capable of running on Windows, Linux and Mac OS X. Please refer to the appropriate part of the following notes on *Installing the Ecere SDK* for detailed installation instructions specific to the platform on which you would like to start developing software. The following chapters assume the SDK is properly installed.

# Section 1

## Programming in eC

A very first program

Variables and data types

Arithmetic, relational, logical and bitwise operators

Flow control

Structures

Enumerations

Arrays, pointers and memory

### *A very first program*

A thousand miles journey starts with a single step.
*-- Tao Te Ching, verse 64*

To keep with the tradition set forth by Brian W. Kernighan, co-author of the book *The C Programming Language*, and because small incremental goals are the key to rapid progress, our first attempt at controlling the computer will be to make it display the following text to a console:

```
hello, world
```

However simple and useless that may seem, we will learn a lot in the process as we familiarize ourselves with a software development environment.

Although the vastness of possibilities they offer us often conceals the reality, computers are in essence machines simply executing specific instructions. The actual set of instructions being executed is dependent on the architecture of the Central Processing Unit (CPU) of the computer on which to run them. The most common architecture family today in personal computers is the Intel x86 series of processors. The internal representation of these instructions, as of any data in a computer, is in binary format (0 - off or 1 - on).

Computer programs are a collection of such instructions to be executed in an orderly manner as to serve a specific purpose. To facilitate the edition and organization of a program, a text representation in a particular programming language is typically used. An assembly language maps one to one each CPU instruction to its textual name. An assembler is used to convert a program written in assembly into executable code.

But what exactly is an instruction? And how can we build complex applications such as a 3D video game out of them? Each instruction by itself accomplishes an extremely specific task. It is the unrestricted ways of arranging them which gives the flexibility to create complex software.

Many instructions perform arithmetic operations. Others simply provide way of transferring data between the CPU registers and system memory. Combining a few of these, we can produce output on devices such as display monitors, printers or receive input from devices such as keyboards or mice. We can transfer data across networks or produce sound on an audio system.

Although assembly programming gives the programmer the ultimate control on how instructions are to be executed, and therefore can theoretically claim the ultimate performance, it has some inconveniences which called for the development of higher level programming languages. One of such a drawback is its inherent association to the computer architecture: assembly is not portable. Furthermore, even though some people are very keen of assembly, and very efficient in writing applications entirely in it, most programmers will find themselves more comfortable and more productive with a higher level language.

Of the thousands of programming languages which have been designed, one of the most influential and enduring is the C programming language, designed by Dennis Ritchie in 1972 for use with the UNIX operating system. C is a purely procedural language, devoid of any object oriented philosophy. However, C has inspired many other object oriented languages such as Objective C, C++, Java, C#. eC (The e stands for Ecere) is no exception.

C is great because it is portable, yet enables a fine level of access to the system memory. This allows for C to be very efficient in terms of program size, memory usage and runtime performance. In this respect, C could be thought of as the next best thing to assembly programming. C was designed as a system programming language, and most operating systems cores today are still written in a mixture of C and assembly. C comes with the C standard library, a standard set of functionality covering many aspects of interacting with the hardware and operating system such as file access, memory management, input/output and mathematical operations.

The Ecere philosophy of programming fully recognizes and embraces the power of C. As such, the eC language is derived from C and is highly compatible with it, more so than most other derivatives. What eC adds to C is a remarkable touch of elegance and simplicity along with object oriented characteristics. It strives to keep its C roots intact.

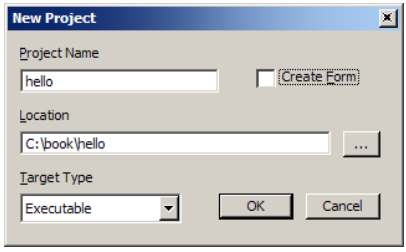
Building an eC, C or C++ application is very similar. First, the program is written using a text editor and saved as ASCII text. A specific extension is associated with each language (.ec for eC; .c for C; .cpp, .cxx, .cc for C++). Then, a *compiler* is used to compile the source code into object code. As a final step, a *linker* builds an executable file out of one or more object file. To simplify this process, a "makefile" containing rules specifying how to perform these actions is typically used alongside a "make" program. Nowadays, the entire process can be done with the help of an Integrated Development Environment which presents an intuitive interface to editing, building and debugging applications. However, it is still common place for UNIX programmers to stick with basic text editors such as *vi* or *emacs* and makefiles.

Before the widespread appearance of graphical user interfaces, consoles receiving characters input from a keyboard, and displaying text output on a display monitor were the primary source of user interaction with computer programs. Multiple terminals providing such console interfaces were often attached to a central computer (which tended to require a lot of space). Although diminishing in popularity, console interfaces can still be found today on most operating systems along with a *shell* in which one can perform system administration tasks, explore file structures or launch applications by typing in their location.

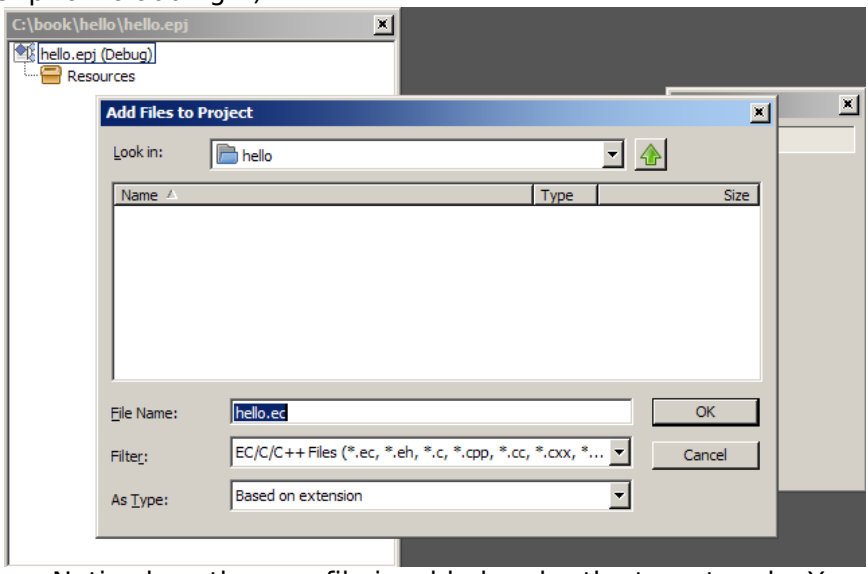
Programming console applications such as the "hello, world" program we are about to write can reasonably be thought to be simpler than developing GUI applications (although you will realize this to be a lot less true with the Ecere SDK, which greatly simplifies GUI programming, than it is with e.g. developing GUIs in C++). Our first applications will thus be interfacing only through the console. Don't worry however, it won't be too long before we delve into building more attractive applications.

Let's start! First, launch the Ecere IDE. To be able to build any application, we will require a project. Let's create a new project: using the menu bar's Project, New we get the following dialog:




We need to provide a project name, and a location for it. Avoiding spaces and sticking to ASCII characters for the project name is recommended. Use a new folder for it, making sure you have the right permissions in the parent folder to create it. The IDE will create the folder if it does not exist. Sticking to ASCII characters for the project location is also a good idea, but spaces there will be fine. We're building a console only application so we don't require a form, let's take out the check mark from Create Form. The target type specifies whether we want to build an executable application or a library to be used as a component part of another application. In our case we want to build an executable. After pressing OK, our project will be ready to use.

Now we can start adding source files to our project. We will call our only file "hello.ec". To do so, locate the target node (hello.epj) in the workspace view (shortcut key: Alt+0), and either right click on it and select "Add Files to Project", or simply press enter while it is selected. Now type in the name of the file to add (it does not need to exist prior to adding it).




Notice how the new file is added under the target node. You can now start editing it by either double clicking on the file or pressing enter while it is selected. Let's type our first eC program[1]:

```
class HelloApp : Application
{
   void Main()
   {
      printf("hello, world");
   }
}
```

Notice how some words come are colored in blue. These are eC keywords; they have a special meaning as part of the eC syntax. eC inherits all C keywords, and adds a few of its own keywords (we will learn later how to work around this as a possible C compatibility issue). Here void is also a C keyword, but class is specific to eC (although it is also a keyword in other OO languages). We will use the same syntax highlighting scheme as the IDE for code samples throughout the book.

1 The equivalent C program follows. Notice the uppercase Main in eC versus main in C.

```
#include <stdio.h>
int main()
{
   printf("hello, world");
   return 0;
}
```

Please notice the indentation of the blocks of code. eC, just like C, is a block structured programming language. Blocks are delimited by curly braces: { and }, and serve to delimit specific constructs such as classes and functions. It is strongly suggested that you follow the same coding conventions when writing your own eC code, which we will lay out throughout the first chapters. Although not mandatory as part of the eC or C syntax (unlike Python where blocks are delimited by indentation), all blocks will be indented by 3 actual space characters per indentation level. Here "void Main()" is indented by 3 spaces, and the "printf" function call is indented by 6 spaces.

These very few lines of code actually expose many fundamental programming concepts which we will take the time to individually analyze. Grammatically speaking, both eC and C are made of what is called "definitions". Definitions are the basic building blocks. In both eC and C, definitions can be broadly categorized as types, variables, and functions. In eC, types occupy a greater importance than in C, and are seen to include a lot more code as object oriented concepts are incorporated into the language.

Indeed, "classes" are a kind of data types which can perform "methods". The next section on object oriented programming will go into much further details. For now, all we need to understand is that we are defining an Application class, HelloApp, and its Main method. The Application class is particular, because its Main method is the *entry point* of an eC application.[2]

The span of the Main method is determined by its block defined with curly braces. Similarly, the definition block of the HelloApp class is defined by curly braces as well.

But what exactly is a method? In procedural programming, we talk about functions. A method is a special kind of function and we will learn about what this means in relation to object oriented programming in later chapters. For now, let's consider this Main method as if it was a normal function, or procedure.

A function in programming is similar to a program at a smaller scale, in that it has an entry and exit point, and executes instructions in between. In fact our program can be thought of as being the Main function, as that is its entry point. When our Main method has finished executing, the program execution will stop as well (the application will terminate). In eC and C, a function consists of a block of multiple *declarations* (we will learn more about these in the next chapter) and *statements*. One thing a statement can do is *call* a specific function.

2 In fact, only one Application class can be active in a single program (defining more than one within the main executable itself yields undefined behavior in regards to which one will be active; Application classes defined in shared libraries will only be active if none is defined in the main executable).

In higher level programming languages, we typically don't deal with individual instructions but rather with bigger building blocks, such as "statements", which can trigger multiple instructions. Since line breaks and spaces in C and eC are only used to separate keywords or identifiers, a special character is used to distinguish multiple statements, the semicolon. It must be added to the end of every statement.

Our *Main* function here consists of a single statement. This statement invokes the *printf* function, which is part of the standard C library. *printf* is said to be an *identifier*. In eC and C, an identifier starts with a letter and contains only alphanumeric characters or the underscore (_) symbol. It is important to note that both eC and C are *case sensitive*. Language keywords cannot be used as identifiers. Identifiers are used to refer to a specific definition, such as a variable or function. When an identifier is used to name a data type, the identifier used becomes a *type* and can no longer be used as an identifier. (More about data types in the next chapter).

It is strongly recommended that you follow the eC conventions for naming identifiers: all functions and data types start with an uppercase, all variables start with a lowercase. *camelCaseVariables* (or *CamelCaseFunction*) are favored to older *underscore_variables* naming schemes. The standard C library functions however all start with a lowercase.

Let's look at that *printf* call in details:

```
printf("hello, world");
```

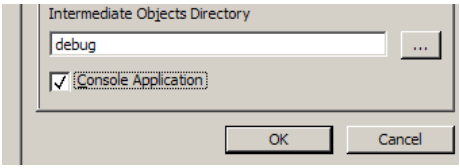
Characters between double quotes form a *character string literal,* or simply a *string*. A string is considered as a whole and forms an *expression*. Functions such as *printf* can be called with *arguments* specifying details on the action it is expected to perform. printf's purpose is to display text on the output of the console. It expects an argument specifying what text to output.[3] When calling a function, a list of arguments in between parentheses must be given, each separated by a comma. One expression must be given for each argument that particular function expects.

We also have the chance to look at the function definition of our *Main* here. Notice the empty parentheses meaning the *Main* function does not take any argument. The void keyword preceding *Main* is the return type of the function. void is a special data type meaning the function does not return any value.

3 The 'f' in printf stands for "format" and refers to printf's highly customizable formatting capabilities. We will discuss these in the next chapter.

Now that we are done writing our program, and we have gotten a good grasp on how it works, let's try building and running it. For console applications a particular care is required if you are running on a Windows environment. Access the Project Settings (shortcut key: Alt-F7, or under the Project menu) and make sure you select the "Console Application" checkbox.




First, try building the application. Select from the menu bar Project, Build (shortcut key – F7). If everything is configured correctly, you should get the following output in the build output tab:

```
Building project hello using the Debug configuration...
Generating symbols...
hello.ec
Compiling...
hello.ec
hello.c
Writing symbol loader...
hello.main.ec
hello.main.ec
hello.main.c
Linking...

hello.exe (Debug) - no error, no warning
```

If you are not getting this but errors instead, the Ecere SDK might not be installed properly. Please refer to the installation notes again. If you are getting syntax errors, you might have mistyped the program. Here is the unfortunate result of forgetting the semicolon:

```
Building project hello using the Debug configuration...
Compiling...
hello.ec
   hello.ec:6:4: error: syntax error
   hello.ec:6:4: error: syntax error

hello.exe (Debug) - 2 errors, no warning
```

Double clicking or pressing enter on the error line in the build output view will bring you directly to the offending line of code. If everything went right, you should now have built your first eC program. Now bring up a terminal window (Command Prompt on Windows) and run the executable from the shell. If you built using the default debug configuration, you would do so by typing "debug/hello" at your shell from within the project's directory. The output should look like this:

```
C:\Book\hello>debug\hello
hello, world
```

### *Variables and data types*

The world is formed from the void,
like utensils from a block of wood.
The Master knows the utensils,
yet keeps to the block:
thus she can use all things.
*-- Tao Te Ching, verse 28*

Programming is much about handling information, or *data*. Information can take many forms: text, numbers, sounds, pictures... Surprisingly, virtually all kinds of information can be be represented and therefore stored as numeric values, to various levels of fidelity. The process of converting otherwise non-numeric information into a numeric representation is known as *digitalization*. Computer programs deal with information strictly in such a digital format.

Data can be permanently stored on mediums such as hard disk drives and optical disks. But to operate with it, applications require the information to be stored in system memory, and for many instructions, in the processor registers themselves. In eC or C however, we do not deal directly with registers, although it is possible to do so through *inline assembly*, small pieces of assembly language embedded in the eC or C code.

Programming languages therefore must provide a way of accessing this information in memory. eC and C offer many ways to do so as we will learn in the upcoming chapters. For now we will consider the simplest way to do so: *variables*. Variables in programming are similar to variables in algebra, in that they hold a value. However, in programming variables always hold a single value which can be known simply by reading its contents.

A variables takes the form of an identifier, which is declared to be of a specific data type. This identifier is allocated and mapped to by the compiler to a specific memory location, taking up as much space as its data type requires. Let's take a look at the *declaration* of a variable "a", of an integer data type.

```
int a;
```

`int` is a keyword for specifying the integer data type. "a" is the identifier which can now be used to access the value of this newly declared variable.

A variable differs from a *constant,* whose value always remains the same. A variable can be assigned multiple values throughout its life span, and its identifier can be used as an expression which will evaluate to the value it was last assigned.

Let's remember computers store information in binary form, as a series of 0s and 1s, or *bits*. To facilitate working with this information, bits are regrouped into *byte*s, most often and for all our purposes consisting of 8 bits. Therefore, a byte can store a numeric value ranging from 0 to $2^8$-1, or 0 – 255. In both C and eC, the char keyword is used to specify a byte data type. The name of the char keyword comes from "character" as characters have long been stored as 8 bit values, potentially allowing for 256 different characters. The ASCII standard requires 7 bits and can represent 128 different characters.

However, by default data types are *signed,* meaning they can have negative values. This affects the range of the data types, as one bit must be sacrificed for identifying negative values. For example, for a char there is still 256 possibles values, but they range from -128 to 127. To specify a data type is to be strictly positive, the unsigned keyword can be used (signed is also a keyword, although rarely used since it is the default). eC adds a built in data type for unsigned bytes: byte. Amounts of memory are usually measured in bytes, or a multiple such as kilobytes (1024 bytes) or megabytes (1024 x 1024 bytes).

This table illustrates the various C and eC integer data types:

| *C type (GCC)* | *eC type* | *bits* | *bytes* | *range* |
|---|---|---|---|---|
| char | char | 8 | 1 (byte) | -128...127 |
| unsigned char | byte | 8 | 1 (byte) | 0...255 |
| short | short | 16 | 2 (word) | -32768...32767 |
| unsigned short | uint16 | 16 | 2 (word) | 0...65535 |
| int | int | 32 | 4 (double word) | $-2^{31}$...$2^{31}$-1 |
| unsigned int | uint | 32 | 4 (double word) | 0... $2^{32}$-1 |
| long long | int64 | 64 | 8 (quadruple word) | $-2^{63}$...$2^{63}$-1 |
| unsigned long long | uint64 | 64 | 8 (quadruple word) | 0...$2^{64}$-1 |

In addition to integers, C and eC offer floating-point data types which can be used to represent real numbers. Two types of floating-point can be used, for different precision and range needs:

| *type* | *bits* | *bytes* | *minimum precision* | *range* |
|---|---|---|---|---|
| float | 32 | 4 | 6 decimal digits | 1.1754943508222875e-38...<br>3.4028234663852886e+38 |
| double | 64 | 8 | 10 decimal digits | 2.2250738585072014e-308...<br>1.7976931348623158e+308 |

A last basic data type exists: void, meaning no data type.

Now that we have a place to store information, we need to examine how to describe it. C and eC offers various ways of doing so, we will take a look at a few of them. Representing data in these ways make up *constant* expressions. Any expression can be assigned to a non-constant expression representing a specific storage place, called in C a *modifiable L-value*. A variable is such an expression.

The most obvious is the integer decimal notation. To assign the number 1234 to the variable "a" previously declared, we simply do:

```
a = 1234;
```

This is an *expression statement*, just like our call to the printf function in our hello, world program. Remember, all statements end with a semicolon. The = symbol is an operator, operating on two expressions, a and 1234. The 1234 expression is a constant. The operator and its two operands together form an *assignment expression* After this statement is executed, the variable a now holds the value 1234.

In C and eC, statements can only be written inside function blocks. The actual function body, enclosed by brackets is itself a *compound statement*. This also means that compound statements can be written inside another compound statement.

Declarations can be written at the global scope, to declare a *global variable,* whose scope is the entire source file, if declared as static[4], or the whole module otherwise. A declaration can be part of structures and classes as we will see in later chapters. They can also be written inside a compound statement, declaring a *local variable,* whose scope is limited to the block in which it is declared.

In eC, declarations and statements cannot be interleaved. That follows the C standard C89, although it is no longer the case with C99. Therefore all declarations must figure at the beginning of a compound block, and no declaration can follow the first statement. However, it is still possible to declare a variable later inside a newly opened compound statement, but its scope will be limited to that new block. eC decides not to allow interleaved declarations and statements for the sake of regrouping and easily spotting every variables which will be used within a particular block. To find the declaration of a particular variable, one knows it will be found a the beginning of the block if it is a local variable. According to the Ecere philosophy, this is more elegant and enhances readability.

4 We will cover the usage of the static keyword and other declarations mode in the chapter about encapsulation and access control.

Let's now put our experiments with variables in perspective. We will insert it inside an application class so we can compile it. We will declare our variable a as a local variable. Remember, statements can only exist within functions. We will use the Main method of our Application class. Consider the following code sample:

```
class VariablesApp : Application
{
   void Main()
   {
      int a;
      a = 1234;
   }
}
```

It is now possible to compile and execute this code. Remember you need to write it in a file with a .ec extension and add it to a project in order to compile it with the Ecere IDE. In order to save space following code snippets will only include the contents of the block. Now, say we want to add a second variable, named b. As we learned, we cannot declare b after the assignment statement. However, multiple *declarators* can be used to declare multiple variables of the same data types in a single declaration. Here we also assign the value of a to this new variable b:

```
int a, b;
a = 1234;
b = a;
```

Now b also holds the same value as a, 1234.

Let's try working with a floating-point variable, to store a fractional number, say 1.618.

```
double phi = 1.618;
```

Notice here how we assigned the value right inside the declaration. Each declarator can have an *initializer*. The decimal notation for fractional number is quite straightforward. It also supports scientific notation such as 1.234e-10. The number 0.5 can also be written omitting the 0, simply as .5. To mark a floating-point number as using single precision, a lowercase f is added to the end of it, like this:

```
float phi = 1.618f;
```

The trailing 0s can also be omitted as in 2. (double precision) or 2.f (single precision) to represent 2.0.

Another important notation is hexadecimal system. In hexadecimal, 6 letters (a – f) are used as digits in addition to the 10 decimal digits, for a total of 16 digits (base 16). Each digit of an hexadecimal number represents a different power of 16, just like in decimal each digit represents a different power of 10.

In eC and C, hexadecimal numbers are preceded by 0x, and the following letters can either be lowercase or uppercase. 0x12AB represents the number 4779 in decimal ($1 \times 16^3 + 2 \times 16^2 + 10 \times 16^1 + 11 \times 16^0$).

In a similar manner, C and eC support an octal notation, which is preceded by a 0. Thus 0123 represents the decimal number 83.

Finally, the last way of specifying values we will look at is characters. A character is enclosed in between single quotes, such as 'A'. The numeric value it represents correspond to the value the character is encoded in. The C language typically supports only ASCII encoding (characters range from 0 – 127). eC files and the Ecere IDE support UTF-8 encoding (a backward compatible superset of ASCII), although these should currently be restricted to the *characters* (between single quotes) and the contents of *string literals* (within double quotes). As such, eC allows any Unicode character within single quotes, and characters which are not part of the ASCII set should be used with the unichar data type built in eC. The unichar data type is unsigned and occupies 32 bit, and is therefore compatible with the uint eC type. Consider the following examples:

```
char a = 'A';
unichar tao = '道';
```

The numeric value of a is now 65, the ASCII mapping of uppercase roman letter A. The numeric value of tao is now 36947 (0x9053), the Unicode code point of the Chinese character 道.

While we are mentioning characters, we will talk very briefly about text strings which will be discussed in a much greater depth later. Text strings are a *string* or series of characters following each other. In C and eC they are enclosed in double quotes. We first used them to say hello to the world in our first program, passing a string as an argument to the printf function. A string literal (a string constant) can be assigned to a variable as follows:

```
char * hello = "hello, world";
```

The asterisk here means hello is a *pointer* to data in the char data type. It points to a string of characters. Pointers is by far the most difficult concept to grasp for beginning C programmers and will be covered thoroughly in the last chapter of this section.

Fortunately, most eC programming does not require dealing with pointers directly, and for now all we need to understand is that a string is declared as a "`char *`". It might be important to note however that the * belongs to the declarator, and not to the declaration specifiers. This means that to declare multiple strings in a single declaration, the * must be repeated:

```
char * h = "hello", * w = "world";
```

We've now covered all the basic data types and how to assign values in various notations to variables declared in these data types. Another valuable concept is the ability to convert one data type into an other, called *casting*. It is implicitly done when assigning one data type to another one, if at all possible. However doing so might result in a warning from the compiler if there is a possible loss of data, such as when converting a floating-point to an integer, or converting a 32 bit integer into a 16 bit integer which cannot hold the entire possible range of the value it is being assigned.

For this reason (and others you will discover while learning eC), a cast operator exists, which explicitly tells the compiler that we wish to perform the conversion. The operator consists of the data type we are converting to between parentheses. For example here is how we would transfer the value of a 32 bit integer into a 16 bit integer:

```
int a = 312;
short b = (short)a;
```

We know the value inside *a* fits within the range of a short, but the compiler assumes *a* could hold any integer value. Here we inform the compiler that this is really what we would like to do. If the value we are casting into a short would not fit within its range, a wrap around would occur (only the bits which would fit would be used). Particular care must be paid to casting signed data types: the most significant bit always being used for the the sign, it differs from types of different sizes. Conversion between signed and unsigned types must also be performed cautiously. Conversion from floating-point numbers to integer numbers result in truncating the number to keep only the integer part. It does not round the number.

```
float phi = 1.618;
int p = (int)phi; // p will be equal to 1
```

Notice our first usage of a *comment*. Comments are for providing insightful information embedded in source code and are ignored by the compiler. This is a C++ style comment which starts with a double slash and will make the rest of the line a comment. eC also supports C style comments which start with /* and ends with */. C style comments can span multiple lines of code.

So far, if you've run the code samples in this chapter, you might have realized that it is not possible to sense their effect. Indeed, they do not provide any output, and most readers won't be able to visualize the changing bits in their system memory just by looking at the silicone chips making it up. In a moment we'll produce some useful output and see the value of our variables change.

It is important to note that a conversion from a character to an integer produces as a result the encoding value, and not the value of a character which so happens to commonly represent a number. For example, character '1' converted to an int is 49, not 1. Conversely, the number 1 converted to a char does not represent the character '1', but the non printable character with ASCII code 1.

Similarly, a character string converted to an integer does not result in the decimal number it may represent. "1234" does not convert into the number 1234, but rather in the address in memory where the string is located (because a string is a pointer, more about this later). An integer cannot be converted to a string simply by casting it either, this is likely to result in the application crashing if we ever try to use this newly casted string.

How then, will you ask, can we convert our variable for presenting it on a text display such as a console? Remember printf from the hello, world of last chapter? We need to learn a bit more about the possibilities it offers. This wonderful function formats text strings by allowing to insert inside the *format string* values of multiple data types. It does so by replacing special formatting characters by additional parameters. Formatting characters start with the % character. Zero or more additional parameters can be passed along, following the format string. It is important that the number of additional parameters matches the expectations of the format string. The data type of each parameter is specified by the formatting characters, in addition to optional specifications of how we would like the data to be converted to text. Let's look at some of the format characters to specify the data type:

| *Format* | *Action* |
|---|---|
| %d | Print a signed integer in decimal notation |
| %u | Print an unsigned integer in decimal notation |
| %x or %X | Print an unsigned integer in hexadecimal notation |
| %f | Print a floating-point number |
| %c | Print the character represented by a byte |
| %s | Print a character string |
| %% | Print a % character (to distinguish from a format char) |

Some examples of format characters which can be used in combination with the above include:

| *Character* | *Example* | *Action* |
|---|---|---|
| [non 0 number] | "%2d" yields " 4" for 4 | Left pad with [number] spaces |
| 0 | "%02d" yields "04" for 4 | If used with a number as above, pad with 0s instead of spaces. |
| .[number] | "%.2f" yields "3.14" for 3.142 | Limit the number of digits after the decimal point. |

[5]

Let's give it a try and print out the values of variables:

```
class VariablesApp : Application
{
   void Main()
   {
      int a, b;
      float phi = 1.618f;
      char comma = ',';
      char * h = "hello", * w = "world";

      printf("The value of a was %d\n", a);
      a = 1234;
      printf("After the assignment, a is now %d\n", a);

      printf("The value of b was %d\n", b);
      b = a;
      printf("After the assignment, b is now %d\n", b);

      printf("phi = %f, more roughly %.1f\n", phi, phi);

      printf("%s%c %s\n", h, comma, w);
   }
}
```

Go ahead and try running this program. What was the value of a and b before they were *initialized*? The answer is *undefined behavior*: it could potentially open gates to Oblivion in your living room. Because local variables are not assigned any particular value until they are initialized, they contain whatever value was at the memory location where they were allocated.

5 This coverage of the printf function is only a quick overview, and is not meant as a reference. For a full description please consult a C reference manual (Try "man 3 printf" on a Linux shell).

I got the following output when running it:

```
The value of a was 31268660
After the assignment, a is now 1234
The value of b was 0
After the assignment, b is now 1234
phi = 1.618000, more roughly 1.6
hello, world
```

The \n which you might have noticed at the end of each string is an *escape character sequence*, indicating to move to the next line of the console. Here is a list of a few useful escape sequences:

| | |
|---|---|
| \n | A new line character (ASCII code 10) |
| \t | A tab character (ASCII code 9) |
| \x[hexadecimal number] | The character corresponding to an hexadecimal number (0x00 - 0xFF) |
| \0[octal number] | The character corresponding to an octal number (or the *null* character if only \0) |
| \" | A double quote character (Useful within " ") |
| \' | A single quote character (useful within ' ') |
| \\ | A backslash character |

A last notion to look at in the context of data types, is that functions can return a value, normally some kind of result of the actions performed within the function. The data type of this value is specified before the function name. As we saw earlier, a void return value means the function does not return a value. A function named Test returning an integer would be defined as such (we will make it return the constant 3 for now):

```
int Test()
{
   return 3;
}
```

We also learned that functions can be passed arguments. Each argument has a data type as well. A function expecting a character string and an integer would be defined like this:

```
void Test(char * s, int i)
{
}
```

A special parameter type allows a variable number of arguments, using the ellipsis symbol (...). The printf function we just used is an example of a function with such arguments. The prototype for printf is the following:

```
int printf(char * format, ...)
```

Note that printf returns a value, but we did not make use of it. Callers are free to ignore a return value. But a call to a function is also an expression, evaluating to the return value of the function. As such, it can be passed as an argument to another function, assigned to a variable, or used anywhere else an expression can be. The following sample illustrates making use of return values:

```
int Test()
{
   return 3;
}

class VariablesApp : Application
{
   void Main()
   {
      int a = Test();
      printf("The value of a is now %d\n", a);
      printf("Test() returned %d\n", Test());
   }
}
```

Note that we chose to declare *Test* as a global function, rather than a method inside VariablesApp. Throughout this first section of the book, we are sticking as much as possible to procedural programming which is mostly valid in standard C, with the exception that we need to define an Application class since it is the entry point of an eC program.

### *Arithmetic, relational, logical, and bitwise operators*

Tao gave birth to One,
One gave birth to Two,
Two gave birth to Three,
Three gave birth to all the myriad things.

*-- Tao Te Ching, verse 42*

A computer's main purpose is obviously to *compute*. They are essentially super calculators. As discussed earlier, most CPU instructions serve to perform various mathematical computations. This chapter will cover both arithmetic and logic operations.

We already learned about the regular assignment operator '='. Similarly, there are C operators to perform arithmetic calculations, some to perform comparisons and others to perform logical operations. We will also learn about some operators which perform arithmetic and assignments simultaneously. Let's remember that operators operate on expressions, which become the *operands* for the operator. Each C operator requires a defined number of operands, either one (unary operator) or two (binary operator), and a special operator which we will take a look at in a later chapter takes three (the ternary conditional operator).

Let's start with the most common arithmetic operators: addition (+), subtraction (-), multiplication (*) and division (/). A few examples follow:

```
int a = 3, b = 2, c;
c = a + b;        // c will be 5
c = a – b;        // c will be 1
c = a * b;        // c will be 6
c = a / b;        // c will be 1
c = a + b – 2;    // c will be 3
```

Note that the result of an operation between two variable of the same data types is of that data type as well. There may be a loss of significant data if the value cannot fit inside that type. For example, the result of an integer division will only contain the integer part. Additionally, a result which is either too big or too small for the range, or would be negative but operands are of an unsigned type, will only be mapped to the bits within the size of that data type.

Some operators can also operate on operands of different data types, and in that case the operand with the highest potential for holding the result will qualify the resulting value. For example, an addition between a `short` and an `int` produces an `int`, and an `int` multiplied by a `float` produces a `float`.

Just like in arithmetics, operators have a precedence. Thus, / and * have a higher precedence than + and -. Operations of equal precedence, such as + and -, are evaluated from left to right[6]. C also offers the parentheses operators ( and ), which have the highest precedence, to explicitly override the order of operations. Arithmetic operators are evaluated before logical operators, logical operators before comparison operators, and logical operators before assignment operators (including combined assignment – arithmetic operators).

```
int result;
result = 1 + 2 * 5;     // result is now equal to 11
result = 1 + (2 * 5);   // same as above
result = (1 + 2) * 5;   // result is now equal to 15
```

We mentioned earlier that the fractional part of the division of an integer by another is lost when using the / operator. There is an operator to obtain the remainder of such a division, or *modulo* (*also called modulus, but not to be confused with the absolute value or module which can also bear that name*). The symbol for the modulo operator is the percentage sign: %. It requires two integer operands.

```
int result, remainder;
result = 11 / 4;         // result will be 2
remainder = 11 % 4;      // remainder will be 3
```

For these five operators, there exists an equivalent combined assignment operator which simultaneously assigns the result of the operation to the left operand. They are: addition/assignment +=, subtraction/assignment -=, multiplication/assignment *=, division/assignment /= and modulo/assignment %=.

```
int a = 3, b = 4, c = 5;
a += b;   // same as: a = a + b, a will be 7
b -= c;   // same as: b = b - c, b will be -1
c *= 2;   // same as: c = c * 2, c will be 10
```

Additionally, there are increment and decrement operators, which are respectively equivalent to += 1 and -= 1. They are ++ and --, and take a single operand on their *right*. These are *prefix* increment and decrement, as the increment / decrement is performed *before* the resulting expression is evaluated.

```
int a = 3, b;
b = ++a; // a and b will be 4 (same as b = a += 1)
```

6 This is true of most C operators, but assignment operators have a right to left associativity, which means they are performed in right to left order.

The *postfix* version of these two operators also exists, taking an operand on their *left*. The increment / decrement is performed *after* the resulting expression is evaluated[7].

```
int a = 3, b;
b = a++; // a will be 4, but b will be equal to 3
```

It is recommended to use the combined assignment operator whenever appropriate, as it greatly enhances the code readability. It might take a little bit of time for programmers used to a language where such operators are not available to naturally adopt them. When the evaluation of an increment or decrement operator is not used, and thus whether using the postfix or prefix version does not make any difference, the preferred Ecere standard is to use the postfix version. In truth such a case is equivalent to += 1, and one might look at it from two different angles. The functional aspect of the += operator is that it behaves like the prefix operator, whereas the syntactical aspect of it is that the expression to be incremented is on the left side of the operator, as it is in the postfix increment operator. In such a situation, the postfix operator is (or was at one point) a more popular choice, as the name of the C++ language might demonstrate.

```
int a = 0;
a++;  /* preferred to ++a, the assignment expression
         value is unused */
```

In addition to performing arithmetic, CPU instructions make it possible to compare two different values. The comparison (or relational) operators include the equality (==), inequality (!=), greater than (>), smaller than (<), greater or equal (>=), and smaller or equal (<=).

The result of such a comparison is either *true* or it is *false*. A true or false value is also called a *boolean* value. C has no specific type for a boolean, but eC defines the type `bool`, which is an *enumeration* with two possible values: `true` (1) and `false` (0). We will learn more about eC enumerations later, but for now know that true and false are valid identifier if they are used with a `bool` data type. Examples follow:

```
bool result;
int a = 3;
result = a > 4;   // result will be false
result = a <= 3;  // result will be true
result = a == 3;  // result will be true
result = a != 3;  // result will be false
```

7 Note that the precedence of the postfix version is higher than the precedence of the prefix version. Both operators are rarely used on the same expression without parentheses to explicitly dictate the desired precedence so the implications are beyond the scope of this chapter.

It is very important not to confuse the assignment operator '=' with the comparison operator '==='. It is a common source of errors for beginners, especially those with previous experience in another programming language such as BASIC, where '=' is for both assignment and comparing for equality; or Pascal, where '=' is for comparison and ':=' is for assignment.

Having two separate operators allows for an *assignment expression* to be an expression, and not a statement such as in BASIC. Every expression evaluates to a value, and in the case of an assignment expression, it evaluates to the value it is being assigned. Consider how the two following statements differ:

```
int a, b;
a = b = 5; // Now both a and b will be 5
a = b == 5; // b was 5, therefore a will be 1 (true)
```

Most computer programs are full of logical evaluations, which control the flow of execution as we will learn in next chapter. All logical operators and flow control statements work with boolean values. Every basic data type we've seen so far can implicitly be converted to a boolean type. The conversion is very simple, if it is 0 (that implies that *every* bit of the data type is set to 0), it becomes *false*, if it is non-zero (if *any* bit of the data type is not a zero), it becomes *true*.

Theoretically, a boolean value takes up a single bit, but because `int` is the *default* data type, eC stores the `bool` data type as an `int`. In order to convert a non boolean value into a boolean value, making sure it is reduced to a single bit to check for its non zero status, you can use the inequality operator along with the 0 constant, as in the following example. Doing so is equivalent to ORing all the bits of the value together (we will learn about the OR operation in just a second). An eC `bool` should always have all of its bits set to 0 except the least significant bit representing the boolean value.

```
int a = 25;
bool result = a != 0;   // result will be true
```

Logical operators are very useful for combining multiple comparisons together, they include the AND operator (&&), OR (||) and NOT (!). Logical operators operate strictly with boolean values. A proper implicit conversion to a bool (a reduction as described above) is automatically performed on each side of the operator before performing the logical operation. This is in contrast with their *bitwise* counterparts we will talk about next, which do not perform this reduction. What this means is that the above check for inequality with 0 is *not required* when using logical operators (which also always evaluate to a boolean value), neither is it with flow control statements which expect a boolean value.

Some programming styles often insist on still putting this inequality, especially when dealing with pointer types. The Ecere coding standard however discourages such usage, for the sake of simplifying expressions and improving readability. Logical operators produce a unique result for a given set of operands, as described in their truth tables:

| AND (&&) | false | true |
|---|---|---|
| **false** | false | false |
| **true** | false | true |

| OR (\|\|) | false | true |
|---|---|---|
| **false** | false | true |
| **true** | true | true |

| NOT (!) | |
|---|---|
| **false** | true |
| **true** | false |

```
int a = 3, b = 0;
bool result;
result = a > 0 && a < 5;   // result will be true
result = a < 0 || a > 100; // result will be false
result = a || b; // true (same as a != 0 || b != 0)
result = !a; // false (same as a == 0)
result = !b; // true (same as b == 0)
```

Remember that operations of equal precedence, such as AND and OR, are evaluated from left to right. To clarify the intent, parentheses should be used to explicitly state the desired precedence of OR or AND when they follow each other, rather than rely on the left to right order. There is a crucial concept to grasp regarding logical operators. In C, Expressions are evaluated in a *lazy* manner. If an inner part of an expression does not require to be evaluated because the logic already dictates the result of an outer expression, those inner expressions are not evaluated. This has very important consequences, as it affects the flow control of a C or an eC program. For example, a function inside such an expression would never be called if it will have no effect on the result of the expression. Consider this example:

```
bool Test(int value)
{
   printf("Testing value %d\n", value);
   return value < 100;
}

int a = 3, b = 4, c = 5;
bool result;
result = a > 10 && Test(b);    // Test will not be called
result = Test(b) && a > 10;    // Test will be called
result = a != 10 && Test(c);   // Test will be called
result = a > 10 && (a = 5);    // a will not be modified
result = (a = 5) && a > 10;    // a will be assigned 5
result = a != 10 && (a = 5);   // a will be assigned 5
```

As mentioned earlier, logical operators perform an implicit conversion to a boolean value. When dealing directly with bits, as it is often useful in slightly advanced programming, that is not appropriate. Therefore operators performing operations on each bits also exist, they are called the *bitwise operators*. There is four bitwise operators dealing with logic: bitwise AND (&), OR (|), XOR (^) and NOT (~). The operation is simply performed on each corresponding bit of the operands one at a time, and go into the corresponding bit of the result.

```
byte a = 3; // binary form:   00000011
byte b = 6; // binary form:   00000110
byte c;
c = a & b;  // c will be 2   (00000010)
c = a | b;  // c will be 7   (00000111)
c = a ^ b;  // c will be 5   (00000101)
c = ~b;     // c will be 249 (11111001)
```

Note that the XOR operator has no non-bitwise equivalent in C, although a non bitwise logical operation can be performed by first explicitly converting to a boolean type and then using the bitwise XOR. Since it didn't figure earlier, here is the XOR truth table. As you can see a XOR operation is true only if a single one of the two operands is true.

| XOR (^) | false | true |
|---|---|---|
| **false** | false | true |
| **true** | true | false |

```
int a = 3, b = 4;
bool result;
result = (a != 0) ^ (b != 0); // Non bitwise use of XOR
```

In addition to these logic bitwise operators, there are bitwise operators to shift the bits. The bits are simply displaced by the number of bits specified by the right operand. The left shift operator (<<) shifts the bits towards the more significant bits, whereas the right shift (>>) shifts the right towards the least significant bit. When shifting left, the least significant bit becomes 0, and the most significant bit which can be hold in the data type is lost. Similarly, when shifting right, the most significant bit of the data type becomes 0, and the least significant bit is lost. Notice how bit shifting multiplies and divides by powers of 2.

```
byte a = 3; // binary form:   00000011
byte b = 6; // binary form:   00000110
byte c;
c = a << 3; // c will be 24  (00011000)
c = b >> 2; // c will be 1   (00000001)
```

Just like the arithmetic operators, all of the bitwise operators except for the bitwise NOT operator have assignment counterparts: <<=, >>=, &=, |= and ^=.

The precedence the bitwise operators in C is particularly low, right between the non bitwise logical operators and comparison operators. This is particularly confusing when used alongside comparison operators, and therefore it is strongly advised that parentheses be used to ensure of the intent. For example, one who would like to verify if both bits 2 and 3 were set might get confused:

```
byte a = 14;      // binary form:   00001110
bool c;
c = a & 6 == 6;   // c will be 0 (14 & 1)
c = a & (6 == 6); // equivalent to above (c = 0)
c = (a & 6) == 6; // intended: c will be 1 (6 == 6)
```

The following table sums up all the operators we've seen so far, from highest to lowest precedence:

| **Operator** | **Description** |
|---|---|
| ( ) | Parentheses |
| ++ -- <br> ( ) | Postfix Increment / Decrement <br> Function Calls |
| ++ -- <br> + - <br> ! <br> ~ <br> (*type*) | Prefix Increment / Decrement <br> Unary Plus / Minus <br> Logical NOT <br> Bitwise NOT <br> Cast to a specific data type |
| * / % | Multiplication / Division / Modulo |
| + - | Addition / Subtraction |
| << >> | Left Shift / Right Shift |
| < <= <br> > >= | Smaller than / Smaller than or equal to <br> Greater than / Greater than or equal to |
| == != | Equality / Inequality |
| & | Bitwise AND |
| ^ | Bitwise XOR |
| \| | Bitwise OR |
| && | Logical AND |
| \|\| | Logical OR |
| = <br> += -= <br> *= /= %= <br> &= ^= \|= <br> <<= >>= | Assignment <br> Addition / Subtraction & Assignment <br> Multiplication / Division / Modulo & Assignment <br> Bitwise AND / XOR / OR & Assignment <br> Left Shift / Right Shift & Assignment |

***Flow control***

For lack of a better name,
I call it the Tao.
It flows through all things,
inside and outside, and returns
to the origin of all things.

*-- Tao Te Ching, verse 25*

Most useful computer programs do not follow a single line of execution. Often decisions must be made regarding which actions should be taken. It is also common for actions to be repeated with different parameters. Functions can be used to regroup actions commonly executed together, and a function can be called multiple times, thus facilitating code reuse. *Iteration statements* or *loops* allow to repeat a block of code multiple times.

A few different types of statements exist in C and eC to control the program's flow of execution. We already saw one, the `return` statement. The return statement's purpose is two fold: it specifies which value the function shall return as we learned, and it terminates the function call (or jumps to the end of the function). The return statement can therefore be used in functions returning no value (of a void return type), with no argument. However, that usage is discouraged by the Ecere standards, as one common exit point (at the end of the function) greatly enhances the function's readability. Having a single exit point also makes it easier to handle any necessary clean up or final action necessary prior to exiting the function. Thus ideally in a function returning a value, a single return statement is required at the end of the function, with a local variable holding the value to be returned until the end. However, in some cases such as relatively short functions, a single return statement would worsen readability rather than improve it, so this is merely a guideline.

The most classic control statement in programming is the `if` statement. We will look at it in details, along with its corresponding `else` clauses. The purpose of if is to evaluate whether a specific *condition* is true or false. The expression to evaluate immediately follows the if, surrounded by parentheses. We already covered the boolean values, comparisons and logical operators in the previous chapter. They are most useful along with the if statement. Remember that an implicit conversion to a boolean type is performed for the expression boolean control statements (such as if) are evaluating, and therefore a "!= 0" check is not required. A sample evaluation could be:

```
if(a > 2 && b < 5)
```

Again, C and eC are blocks structured languages. Therefore if doesn't have a matching END IF statement such as in BASIC for example. Both if and else must be followed by a statement, which can be a compound statement containing multiple statements. Therefore the if and else clauses end with the end of the statement or compound block. Consider the following examples:

```
if(a > 2 && b < 5)
   printf("this\n");
else
   printf("that\n");
```

is equivalent to

```
if(a > 2 && b < 5)
{
   printf("this\n");
   /* More statements could follow, all be
      executed if the condition is true */
}
else
{
   printf("that\n");
   /* More statements could follow, all be
      executed if the condition is false */
}
```

Often series of conditions must be checked, and if and else find themselves gathered together. C doesn't have a specific ELSE IF statement as other languages such as BASIC do. Instead, an `else if` is actually a combination of an else and an if. It is written this way:

```
if(a > 2)
   printf("this\n");
else if(b < 5)
   printf("that\n");
else
   printf("nothing\n");
```

and is equivalent to:

```
if(a > 2)
   printf("this\n");
else
{
   if(b < 5)
      printf("that\n");
   else
      printf("nothing\n");
}
```

A common ambiguity with the if / else statements of various programming languages is also present in C. It is called the "dangling else". Consider the following:

```
if(a > 2)
   if(b < 5)
      printf("this\n");
else
   printf("that\n");
```

The indentation here suggests "that" will only be printed if a is not greater than 2. In fact, the C grammar dictates that the else here belongs to the `if(b < 5)`, and is thus equivalent to:

```
if(a > 2)
{
   if(b < 5)
      printf("this\n");
   else
      printf("that\n");
}
```

In order to avoid this ambiguity when reading the code, as well as ensuring our intentions, any if or else statement containing another if statement should explicitly contain a brackets enclosed compound statement. So what we intended to write here (again, according to the way we indented the example) was:

```
if(a > 2)
{
   if(b < 5)
      printf("this\n");
}
else
   printf("that\n");
```

The only exception is the particular else if statements as described earlier, which should also figure at the same indentation level, with else and if on the same line, The C grammar dictates these to work like a typical "else if" construct. Any if statement within an else if construct which is not part of the else if series however, should be contained within a compound block. Therefore the following is wrong:

```
if(a > 2)
   printf("this\n");
else if(b < 5)
   if(a > 0)
      printf("that\n");
else
   printf("nothing");
```

It should have been written as:

```
if(a > 2)
   printf("this\n");
else if(b < 5)
{
   if(a > 0)
      printf("that\n");
}
else
   printf("nothing");
```

By now you should have started to realize the importance of properly indented code. This sums up the `if` statement, and we will now take a look at the other C selection statement: `switch`. The `switch` statement is closely related to the `else if` statement, as it could be implemented as such. Similarly, a series of else if statements comparing the same expression for equality to different values can be (and should be) implemented as a switch statement. Consider this example:

```
int a = 2;
     if(a == 1) printf("One");
else if(a == 2) printf("Two");
else if(a == 3) printf("Three");
```
[8]

The switch statement allows us to specify the expression to be written and evaluated only once and rewrite it as such:

```
int a = 2;
switch(a)
{
   case 1: printf("One");   break;
   case 2: printf("Two");   break;
   case 3: printf("Three"); break;
}
```

Evaluating only once is particularly useful if the expression is more complex than a single variable or needs to perform work such as invoking a function call. In the case of the switch statement, the function will only be called once. With else if statements it would be called multiple times, unless the function is called before the else if statements and its return value saved for the purpose of evaluation.

8 Notice that I took the liberty to put each if and else if statement on a single line. One of the Ecere principles for code readability is to be able to see as much of the code as possible at once, in order to get a better global picture. Therefore for such short lines of code, you will often see regrouping like this resulting in fewer lines of code which improve readability and save space. Especially with the advent of cheaper huge high resolution wide screen monitors, the 80 console characters limit is obsolete, and fewer, longer lines are preferred.

The case statement is used only in pair with switch, which is why it's often referred to as the switch/case statement. The case statement takes in a single argument which must be constant. Multiple case statements with the same values within the same switch are not allowed. Unfortunately, it is not possible to evaluate comparisons other than equality within a switch statement, as it is in BASIC for example. One must therefore resort to else if statements to perform evaluations including greater than or smaller than operators.

The break statement has a particular meaning within a switch statement, but will also be used with iteration statements. Within a switch, it simply signifies to stop executing the statements within the switch. Indeed, if a break is not present, the statements of the following case statements will be executed as well. This is not obvious at first, and it is a common source of errors to forget the break statement. For example:

```
int a = 2;
switch(a)
{
   case 1: printf("One");
   case 2: printf("Two");
   case 3: printf("Three");
}
```

This will print both "Two" and "Three". In some cases, this might be the desired behavior and the break statement is voluntarily omitted in order to perform a common action for two different values, but one additional action (performed before the common action) for one of these two values. Note that the values need not be in ascending order, but can be listed in whichever way appropriate to the situation.

Now what if none of the cases matches the value? In the above examples, nothing; none of the case statement within the switch is executed. However C provides an additional keyword for specifying what is to happen if no case matches: `default`. Example:

```
int a = 4;
switch(a)
{
   case 1:  printf("One");   break;
   case 2:  printf("Two");   break;
   case 3:  printf("Three"); break;
   default: printf("Neither One, Two nor Three");
}
```

The default statement should always be the very last. Just like with case, the above case statements will go into the default statements if the break statements are omitted.

Compound statements can also be used within case statements in order to define variables. More complex switch statement will look like this: (take note of the indenting which follows the Ecere standards)

```
int a = 4;
switch(a)
{
   case 1:
   {
      int variable;     // Sample declaration
      printf("One");
      break;
   }
   case 2:
   {
      printf("Two");
      break;
   }
   case 3:
   {
      printf("Three");
      break;
   }
   default:
   {
      printf("Neither One, Two nor Three");
   }
}
```

Now that we've seen both selection statements, we'll study the iteration statements. There are three iteration statements in C: `while`, `do ... while` (a slight variation of `while`) and `for`. Each of these also perform a kind of selection on a boolean value to decide whether or not to continue iterating. Let's look at the simplest first, `while`.

```
int a = 10;
while(a > 0)
   printf("%d\n", a--);
```

The while statement first checks the boolean status of the expression to be evaluated, then if it is true executes the statement. It repeats until the expression is false. The example above will print numbers 10 to 1 in decreasing order. Notice our use of the postfix decrement operator. Using the prefix version (--a) would have printed the numbers 9 to 0. A compound statement can of course also be used within a while in order to execute multiple statements.

A very slight variation of the while loop is the do ... while. The only difference is that it does not perform an initial evaluation, and thus the statement will be executed at least once, regardless of the value of the expression.

```
int a = 10;
do
{
   printf("%d\n", a--);
} while(a > 0);
```

Note the mandatory semicolon at the end of the while clause of the do ... while statement. This will produce the exact same output as the previous while statement. However, the following example will print a single line with "0" whereas the previous while example would not have printed anything:

```
int a = 0;
do
{
   printf("%d\n", a--);
} while(a > 0);
```

We mentioned earlier that the break statement had a different meaning within a loop. Inside a a while, do ... while or for, break can be used to immediately jump outside the loop. Infinite loops can then be meaningful with hope that they will eventually end:

```
int a = 10;
while(true)
{
   printf("%d\n", a--);
   if(!a) break;  // The loop will end when a is 0
}
```

The break statement only applies to the innermost loop (or switch) within which it resides. Note that this makes it impossible to jump out of a loop from within a switch, because the break will simply end the switch statement. Flag variables must then be set within the switch, which can later be checked once outside the switch where it is possible to exit the loop. The same method can be used to break out of more than one loop at once. The example which follows illustrates such a situation with one while loop within another. Each loop simply increments a variable and do not perform anything useful. Both loops end when the check within the inner loop finds that both a and b are equal to 5.

```
bool end = false;
int a = 0, b = 0;
while(a < 10)
{
   int b = 0;
   while(b < 10)
   {
      if(a == 5 && b == 5)
      {
         end = true; // Set a flag to exit outer loop
         break;      // Break out of inner loop
      }
      b++;
   }
   if(end) break; // Break out of outer loop
   a++;
}
```

All of the loop examples so far iterated the values of variables. That is a very common occurrence in programming to do so, and a special construct for this purpose exists in most languages, the for loop. The C for loop however is more flexible than many other languages. It consists in three parts (in addition to the the statements to be repeated): the *initialization*, the *evaluation* and the action to be performed *after each iteration* (most often an *incrementation*). Each of the three parts is an expression, and they are separated by semicolons. Any of the parts can be omitted (the evaluation defaults to *true*). Our first while loop example can be rewritten as such:

```
int a;
for(a = 10; a > 0; a--)
   printf("%d\n", a);
```

A classical increasing for loop counting from 1 to 10 would be:

```
int a;
for(a = 1; a <= 10; a++)
   printf("%d\n", a);
```

Note that most often if the for loop is to execute something a number of times, the counter will start at 0 rather than 1, and the smaller than operator will be used rather than smaller than or equal to:

```
int a;
for(a = 0; a < 10; a++)
   printf("This will print 10 times.\n");
```

An infinite for loop can be written by omitting all parts. It is equivalent to the `while(true)` loop.

```
for(;;)
   printf("This will never end\n");
```

Note that multiple expressions can be grouped together in a single expression, separated by commas (`,`). In this case, the overall expression evaluates to the *last* expression. This is particularly useful within the for loop, for example to initialize and iterate multiple variables:

```
int a, b;
for(a = 0, b = 10; a < 10; a++, b++)
   printf("a = %d, b = %d\n", a, b);
```

As me mentioned earlier, the `break` statement can also be used within a for. The counterpart of break also exists: `continue`. The continue statement terminates the current iteration, performs the post iteration action (in the case of a for), evaluates the condition and starts a new iteration if the condition is true. It can also be used with `while` and `do` ... `while` loops. The continue statement can be very useful to skip the remaining of the current iteration and move on to the next. Consider the following example:

```
int a;
for(a = 0; a < 10; a++)
{
   if(a == 4) continue;
   printf("a is %d.\n", a);
}
```

The above will print values 0 to 9, excluding 4.

This concludes this chapter on flow control statements. We are deliberately skipping over the C `goto` statement as the Ecere coding standard considers it bad practice, and most uses of the goto statement could be written more elegantly using the great flexibility of the selection and iteration statements we just described. However, eC does support the goto statement.

## *Structures*

The Master views the parts with compassion,
because he understands the whole.

*-- Tao Te Ching, verse 39*

The universe has a natural tendency towards self-organization, of which we are living proof. Our body is made up of organs, themselves made up of tissues, tissues consisting of arrangements of cells, each cell having its own components, built with molecules comprised of atoms.[9]

Just like a human body, a complex program comes to life through a hierarchical organization of simpler parts. We've already learned that eC statements are implemented with CPU instructions, that they can further be organized into functions, and that variables are useful to organize memory. CPU instructions are also stored in memory. Both memory and the execution of instructions are in fact themselves powered by electric energy.

Up until now we've always dealt with basic data types for data, a variable holding a single value, and each variable being unrelated to every other variables. With *structures*, it is possible to organize data into more complex data types, allowing to create any hierarchies of blocks of data which can best represent the information we wish to manipulate. A block of data organized in such complex types can be thought of as an *object,* as it is often mapped to a conception meaningful in the real world or in a context specific to the application. Indeed, although many procedural programming languages such as C do not facilitate support for object oriented programming, structures provide the building blocks for it and thus make it possible (although cumbersome) to write object oriented code in these languages.

In the next section we will discuss object oriented programming in depth, and learn that all eC data types (including the structures we are about to describe) support object oriented features. For now, we will focus on the data storing aspect of structures. Nevertheless the concepts explained in this chapter (which will cover both the standard C structures and the particularities of the eC language) are the very foundation for the object oriented constructs taught later on.

9 *Atom* comes from the Greek *atomos,* meaning "uncuttable". With the advent of quantum physics, scientists realized that matter is actually nothing but a form of energy. Delving deeper, physicists learned that it is impossible for an observer to carry any experiment without affecting the system being observed, and thus the duality between observer and observed, or mind (thought) and matter (energy) is merely a conception of the mind.

Structures in C and eC are defined with the `struct` keyword and list of *data members* which the structured data type will contain. Each data member is qualified with a data type. eC offers a simpler way than C to declare named structured data types which is more in-line with modern programming languages. Consider the following declaration:

```
struct InventoryItem
{
   int code;
   float price;
};
```

An *InventoryItem* data type is declared, containing both a code which is to be stored as an integer, and a price which is to be stored as a floating point number. This new data type will take as much memory as is required for holding all the data.[10] An inventory item variable can then be declared as such:

```
InventoryItem item;
```

Each data member can be accessed individually through the variable identifier and the member operator `'.'` followed by the member identifier:

```
item.code = 1234;
item.price = 45.0f;
```

Structure variables can also be initialized within their declaration through *list initializers* specified within curly brackets:

```
InventoryItem item = { 1234, 45.0f };
```

List initializers are assigned to the data members according to the order of the data member declarations. eC offers another way to write the above which will be referred to as the *instantiation* syntax. An instance (or object) is an object oriented term to refer to a single occurrence of a specific data type (we will cover the subject in a lot more detail in the next section). In the above example, *item* is an instance of the *InventoryItem* structure type. As you can see in the following example, the eC instantiation syntax is greatly inspired from C's list initializers:

```
InventoryItem item { 1234, 45.0f };
```

10 Depending on the *padding* necessary to align data members, this might be more than the sum of the size of the data type of all members.

In addition to not requiring the = sign, the instantiation syntax present a few advantages. Any or all data members can be omitted, and any data member can be explicitly set through its name[11]:

```
// code will be 0
InventoryItem item1 { price = 45.0f };
// price will be 0
InventoryItem item2 { code = 1234 };
// both code and price will be 0
InventoryItem item3 { };
```

Note that using the instantiation syntax guarantees the storage for the structured to be initialized (by default to 0) and thus no member will contain stray garbage data. If neither using the instantiation syntax nor assigning a list initializer, all data members will contain uninitialized memory.

Just like C list initializers, the instantiation syntax follows the data member declaration orders. However whenever a member is specified by name, the next unnamed member to be initialized is reset to the data member following that named initialized member. For example:

```
struct InventoryItem
{
   int code;
   float price;
   float cost;
};
// 20.0f will be assigned to the cost
InventoryItem item { price = 45.0f, 20.0f };
```

This way of declaring a structure type is specific to eC and only allowed at the global level (outside of any function). Structures are a delicate point of the C compatibility of eC. Indeed, by design eC makes every possible effort to support this simpler way of declaring named structure data types, while at the same time maintaining compatibility with standard C structure declarations. The following discussion will attempt to clarify the details pertaining to the use of both, and is primarily intended for previous C programmers and for developers intending to use an external library through the include files (headers) of a C application programming interface (API). New developers can feel free to skim over the next discussion (about a page and a half) rapidly and stick to the eC syntax. First we will study how structures work in standard C, and how eC accommodates both syntaxes.

11 More recent C standards have a similar functionality whereas the "[.member name] = value" syntax can be used within a list initializer to set a specific data member. That syntax is not supported in eC and can be argued to be not as elegant as the eC instantiation syntax which does not require the dot.

In C, a `struct` declaration is actually a data type specifier, just like `int`. Therefore a structure variable can be declared as such:

```
struct { int code; float price; } item = { 1234, 45.0f };
```

This is valid both at the global level and for local variables inside functions (in eC as well). As you can see, the `struct` keyword followed by the data member declarations acts just like a basic data type, and can declare one or more variables, each of which having an optional list initializer. Because a specific structure is typically used more than once, C offers a way to name each structure (with an identifier immediately following the `struct` keyword) so that the list of data members need not be repeated every single time. This however declares a structure, only visible at the scope where it is declared, which cannot be used by itself as a data type. In order to declare a variable, the `struct` keyword must be used again, like such:

```
struct InventoryItem
{
   int code;
   float price;
};
struct InventoryItem item2 = { 5678, 55.0f };
```

Note that declarators and initializers can still be used along with named structures. To further facilitate things, C also provides the `typedef` keyword which enables the creation of new data types. `typedef` maps any existing data type to a new identifier, which can then be used by itself in declarations as a data type. It can be used for example with basic data types:

```
typedef int MyInt;
MyInt a = 5;
```

It can also be used with structures, so that our *InventoryItem* example can be rewritten as:

```
struct InventoryItem
{
   int code;
   float price;
};
typedef struct InventoryItem InventoryItem;

InventoryItem item1 = { 1234, 45.0f };
```

As you can see, the same identifier can be used for the type and the structure, as both are different classes of identifiers (structures and types).

Alternatively, the structure can be declared within the typedef:

```
typedef struct
{
   int code;
   float price;
} InventoryItem;

InventoryItem item1 = { 1234, 45.0f };
```

Note that *InventoryItem* here is the name of the type, not a `struct` declarator. A structure identifier could also be used along with a `typedef`, so that it is possible to declare variables as "`struct` InventoryItem", which would otherwise not be recognized.

```
typedef struct InventoryItem
{
   int code;
   float price;
} InventoryItem;

InventoryItem item1;
struct InventoryItem item2;
```

All these C constructs can also be used within eC code, but note that a global named `struct` is both the eC syntax and a valid C syntax. As such, it can be used both ways. However, if a type with the same name is declared using `typedef`, it cannot be used as an eC `struct` (for example, the instantiation syntax is not available with the last three examples). The following is valid eC code without requiring a `typedef`:

```
struct InventoryItem
{
   int code;
   float price;
};

InventoryItem item1 = { 1234, 45.0f };
struct InventoryItem item2 = { 5678, 55.0f };
InventoryItem item3 { 9012, 65.0f };
```

Another particularity of the eC `struct` is that function parameters declared with its syntax are automatically passed *by reference*. This means that when a function modifies the contents of a `struct` parameter, it is actually modifying the structure variable passed by the caller. This also avoids putting the entire contents of structures on the *stack*, as a C structure would naturally do unless pointers are used. Consider the following example:

```
struct InventoryItem
{
   int code;
   float price;
};

void FillItem(InventoryItem item)
{
   item.code = 1234;
   item.price = 45.0f;
}

void Test()
{
   InventoryItem item1 { };
   FillItem(item1);
   // item1 now contains 1234 and 45.0f
}
```

This is a particularly useful way of returning multiple values from a function through its parameters (as does *FillItem*), since a function can only return a single value from its return data type.

If such a behavior is not desired, it is possible to declare a structure parameter using the C syntax as such:

```
void DontModifyItem(struct InventoryItem item)
{
   // This only modifies a local copy
   item.code = 1234;
   item.price = 45.0f;
}

void Test()
{
   InventoryItem item1 { };
   DontModifyItem(item1);
   // item1 still contain zero values
}
```

eC also supports anonymous instantiations. Whereas a named instantiation is a declaration, an anonymous instantiation is an expression, and as such can be used wherever an expression is required, such as an argument to a function:

```
void Test()
{
   DontModifyItem(InventoryItem { 1234, 45.0f });
}
```

When a specific data type is expected, such as in a function argument, the name of the matching structure can be omitted, allowing for more elegant code:

```
void PrintItem(InventoryItem item)
{
   printf("Code: %d, Price: %.2f\n", item.code,
      item.price);
}

void Test()
{
   PrintItem({ 1234, 45.0f });
}
```

Whether using the C or eC kind, structures can also be nested within each other. For regular data members, the inner structure is simply the data type of the member, and inner members are used through a succession of dots and member identifiers. The '.' member operator's associativity is left to right, and it has the same precedence as the postfix increment/decrement and the function call parentheses.

```
struct Cost
{
   int dollars;   // We decide to use two integers to
   int cents;     // represent the dollars and cents
};

struct InventoryItem
{
   int code;
   Cost price;
};

void Test()
{
   // We can use the . inside the instantiation
   InventoryItem item { price.dollars = 5 };

   // Next line is same as: (item.price).cents = 10;
   item.price.cents = 10;
}
```

It is additionally possible to have anonymous structures within a structure, in which case the extra member can simply be skipped, and the internal members can be accessed directly through the dot operator, as the following sample demonstrates.

```
struct InventoryItem
{
   int code;
   struct
   {
      int dollars;
      int cents;
   };
};
void Test()
{
   InventoryItem item { dollars = 5 };
   item.cents = 10;
}
```

Structures have very close relatives called *unions*. In all aspects a union works exactly like a struct (it also supports eC syntax), except that all its data members start at the same memory emplacement. It is particularly useful for versatile data types which could hold different types of objects (often identified with a type identifier variable), or for accessing the same data in different ways. The size of a union is the size necessary to hold the largest of its data members. The use of unions lies somewhat in advanced programming topics, so for now we will simply present an example of its syntax:

```
union VersatileType
{
   int value;
   InventoryItem item;
};

void Test()
{
   VersatileType data { value = 10 };
   // 10 has been assigned to value, but the
   // InventoryItem's code member which maps to the
   // same address in memory now also contains 10.
   data.item.dollars = 20;
}
```

We've talked about the size of a structure a few times by now; it would be useful to know that the `sizeof` operator exists which returns the size (in number of bytes, always a constant expression) of any data type, computed at compile time. It is also possible to use the `sizeof` operator with a variable, in which case it returns the storage size for that variable's data type. Because of particularities with the `sizeof` operator's precedence and associativity, following the Ecere coding standards requires its argument to always be enclosed within parentheses, such as: `sizeof`(VersatileType).

***Enumerations***

In emptiness, there is no form,
no feeling, thought, activity, consciousness.
No eye, ear, nose, tongue, body, and intellect consciousness.
No form, sound, smell, taste, touch and phenomena.
No realm of eye consciousness
till no realm of intellect consciousness.
No ignorance and no termination of ignorance,
till no age and death and no termination of age and death.

*-- The Prajñaparamita Hrdaya Sutram (Heart Sutra)*

It is common for our minds to conceptualize nature in a dualistic manner, e.g. male, female; light, dark; hot, cold. Each of these however are but two aspects of an interrelationship, whereas one aspect would not be meaningful without the existence of its opposite. The symbol of the Tao illustrates very well the unity of the universe (represented by the circle) through duality, with its interdependent *yin (*passive, dark, female...) and *yang* (active, light, male...) aspects.

Thus a *temperature* could be thought of as either *hot* or *cold*, *sex* could be categorized as being *male* or *female*, and a generic *aspect* could be either *yin* or *yang*. We've already seen that a *boolean* value can be *true* or *false*. These are all examples of binary *enumerations*, which can be one of two possible values. It is also possible to have enumerations with a greater number of possible values. For example Chinese philosophy sees the entire universe as being made of five elements (better translated as phases, not to be confused with chemical elements): *wood* (木), *fire* (火), *earth* (土), *metal* (金) and *water* (水).

As in C, the eC keyword to define an enumeration is `enum`. The syntax is rather simple:

```
enum Element { wood, fire, earth, metal, water };
```

Note that following the Ecere coding standards requires all values to start with a lowercase, whereas the enumeration name (like all other data types) starts with an uppercase. *Element* variables can then be declared and assigned any of the possible values:

```
Element element1 = water, element2 = earth;
```

By default, enumerations represent a value of the `int` data type. In eC only, it is possible to override which data type it will use (which can be any integral data type), by following the enumeration name with a colon (:) and the desired type:

```
enum Element : byte { wood, fire, earth, metal, water };
```

It is also possible to assign a specific value to any of the enumeration values. If none is assigned, the first value starts at 0, and the following ones are incremented by one, in the order they are declared. As an example, say we wanted our elements to start at 1 rather than 0, we could simply do:

```
enum Element { wood = 1, fire, earth, metal, water };
```

And fire, earth, metal and water would respectively be assigned 2, 3, 4 and 5. A particularity of the eC enumerations is that they are context sensitive. We've briefly mentioned earlier that the `true` and `false` identifiers are only valid when used with a `bool` data type. Indeed, the values for any specific enumerations are only recognized when that enumeration is the expected data type of an expression. This differs from standard C where all value identifiers are available in the entire scope of the enumerations.

The advantage of this is that multiple enumerations can use the same identifier without clashing with each other. Say we had another enumeration using the earth value identifier:

```
enum Planet
{
   mercury, venus, earth, mars, jupiter, saturn, uranus,
   neptune
};
```

In standard C these two enumerations could not coexist within the same scope. Furthermore, note that *earth* has a value of 3 in our previous *Elements* example starting at 1, and a value of 2 in our *Planet* enumeration. To make them coexist, we would need to prefix all values by something meaningful to differentiate the two values, say *`elementEarth`* and *`planetEarth`*. But in eC that is not required:

```
Planet planet = earth;   // This will be 2
Element element = earth; // This will be 3
```

However the following is not recognized:

```
int a = earth; // error: Unresolved identifier earth
```

Simply as an example, the built in eC `bool` enumeration would be declared as such:

```
enum bool { false, true };
```

Just like structures, eC has a slightly different, simpler syntax than C for enumerations, but still fully supports the C syntax. We will now take a look a the C syntax for those who might find it useful.

C's enumeration syntax is similar to its approach to structures, in that it doesn't define a new data type by itself. The identifier naming the enumeration indeed merely defines a new enumeration. As for structures, the enum keyword must be used along with the enumeration name to declare a variable. Like structures also, the enumeration itself can be anonymous and be used as a data type.

```
enum Element element = earth;
enum { no, yes } answer = yes;
```

The common usage however is with a typedef, either in a separate declaration, or most often together with the enumeration.

```
typedef enum Element Element;
typedef enum
{
   wood, fire, earth, metal, water
} Element;
```

An *Element* enumeration type will then be available so that the following code is valid:

```
Element element = earth;
```

Just like structures, the eC syntax for enumerations is only available at the global level, whereas the C syntax is available within any compound statements. Remember that using the C syntax value identifiers are available within the entire scope of the enumeration, whereas using the eC syntax identifiers are available according to the expression's expected data type.

An additional feature of eC is the possibility to obtain the size of an enumeration (the largest value + 1), as a compile-time constant value. Here is an example of the syntax with our *Planet* enumeration:

```
int size = Planet::enumSize;
```

A last feature is the ability to *derive* one enumeration from another, adding new possible values. It is done using the syntax we saw earlier for selecting the enumeration's data type, choosing the base enumeration as the data type:

```
enum PlanetOrDwarfPlanet : Planet
{
   ceres, pluto, eris   // ceres will be 8, and so on
};
PlanetOrDwarfPlanet planet1 = mercury;
PlanetOrDwarfPlanet planet2 = pluto;
```

***Arrays, pointers and memory***

Music or the smell of good cooking
may make people stop and enjoy.
But words that point to the Tao
seem monotonous and without flavor.
When you look for it, there is nothing to see.
When you listen for it, there is nothing to hear.
When you use it, it is inexhaustible.

*-- Tao Te Ching, verse 35*

Pointers are one of the most difficult concept to grasp in C programming. Indeed, many modern programming languages opted not to provide direct access to memory, for various reasons such as safety and simplicity. This of course was not an option in the design of eC, which aims for full compatibility with the C language, and strives to achieve maximum runtime performance. However, pointers do add an extra level of complexity, which can be quite a burden in writing simple object oriented application which do not really require a thorough understanding of memory access.

Fortunately, the middle-way approach of eC makes pointers totally transparent until they are truly needed. Thus it is not required to be familiar with pointers to write useful object oriented code. The topic is still presented in this last chapter of the basics of programming in eC, because it does not belong to object oriented programming and is the last topic to be covered which also applies to the C programming language. By the end of this chapter, you will have completed an in-depth introduction to C programming.

Although it is recommended to read it through at least once before moving on to the next section, don't worry if you do not feel you are mastering the subject just yet. Pointers will not be used in the book until we cover text strings and file access in later sections. Coming back to this chapter after having written a bit of code and understanding the purpose of pointers will be most useful.

Before teaching pointers, this chapter will explain arrays which are closely related to pointers. We will also discuss memory allocation and addressing.

In C and eC, an *array* allows to have more than one instance of a data type, allocated in contiguous memory, accessible through an array variable and a numeric *index*. Square brackets are used both to declare the number of elements an array should have, and to specify the index to be used. Indexes go from 0 to the number of elements - 1.

```
int array[10]; // Declare an array of 10 int elements
array[0] = 1; // Set the first element (index 0) to 1
```

Reading from or writing to an index beyond the array's declared size will be accessing memory not belonging to the array and in most cases will result in an undesired behavior, often an illegal memory operation which might terminate the application unexpectedly. For example, the following is wrong:

```
int array[10], a;

array[-1] = 1; // writes to memory before array
array[10] = 1; // writes to memory after array

a = array[-1]; // reads from memory before array
a = array[10]; // reads from memory after array
```

Like normal variables, arrays are not initialized by default (unless they are part of another data type being initialized, such as a structure being instantiated). It is possible to initialize them with a list initializer, whereas each element in the list is associated with an element in the array:

```
int array[10] = { 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 };
```

The first two elements of the array (index 0 and 1) will contain 1, and the following values are assigned to indexes 2 and on, the last element (index 9) being assigned 55. If elements are omitted in the initializer list, they will be assigned to 0. For example:

```
int array[10] = { 1 };
```

The first element (index 0) will be 1 and all other nine elements will be initialized to 0. It is also possible to declare arrays of structures:

```
struct InventoryItem
{
   int code;
   float price;
};
InventoryItem items[3] =
{
   { 1234, 45.0f },
   { 5678, 55.0f },
   { 9012, 60.0f }
};
void PrintItem(int i)
{
   printf("Code: %d, Price: %.2f\n",
      items[i].code, items[i].price);
}
```

Note how the initialization of items is a list initializer themselves containing a list initializer for each item in the array. Also observe that a variable can be used to index an array; any integral type can be used. An array declaration however requires the size specified to be an integer constant. The precedence and associativity for the `[ ]` indexing operator is the same as the function call `( )` or the `.` to access structures' members. Notice how it is used alongside the `.` operator to access the *`code`* and *`price`* member of the *`InventoryItem`* structure elements.

The size in bytes of an array is equal to the product of its number of elements by the size of the data type of each element. The `sizeof` operator can be used with the array variable to obtain it. Thus in the preceding example, `sizeof(items)` is equivalent to `sizeof(InventoryItem) * 3`.

It is also possible to declare multi-dimensional arrays. So far we demonstrated 1-dimension array (with a single index). Although it could be decided arbitrarily which index represents what, the last index typically represents *columns*, whereas the next to last represents *rows*, which matches the way the initializer list can be organized. Although more difficult to visualize, it is possible to have arrays with more dimensions than 2 or 3. Therefore a 2 columns by 3 rows array could be used as such:

```
int array[3][2] =
{
   { 1, 2 },
   { 3, 4 },
   { 5, 6 }
};

void PrintArray()
{
   int j; // row counter
   for(j = 0; j < 3; j++)
   {
      int i; // column counter
      for(i = 0; i < 2; i++)
         printf("%d  ", array[j][i]);
      printf("\n");
   }
}
```

When declaring array parameters of a function, the number of elements for each dimension must be specified, with the highest level (the leftmost brackets, rows in our 2 dimensions example) being optional. Therefore for single dimension arrays it is not necessary to specify any size. We could rewrite our *`PrintArray`* function as such:

```
void PrintArray(int a[][2], int nRows)
{
   int j; // row counter
   for(j = 0; j < nRows; j++)
   {
      int i; // column counter
      for(i = 0; i < 2; i++)
         printf("%d  ", a[j][i]);
      printf("\n");
   }
}
```

It would then be suited for printing any two-dimensional integer arrays with two columns, regardless of the number of rows (which we are now passing as a parameter along with the array to be printed). To print our array above, we sould simply invoke it as such:

```
PrintArray(array, 3);
```

It is also possible to omit the highest level size when declaring an array if it has an initializer list, in which case the number of elements in the initializer list is used to figure out the size:

```
int array[] = { 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 };
```

Here the implied number of elements is 10, and it is thus equivalent to declaring as `int array[10]`.

Like all the data types we have seen so far, the memory for the elements of an array variable is either allocated as part of another allocation if it is a member of another data type (such as a structure), in a special block of memory available throughout the program's existence if they are global variable, or on the *stack*, if it is a local variable (inside a function).

The stack is special memory construct which ensures the variables for a function will remain alive as long as we are within that function or within a deeper function called from there. The variables for that function will cease to exist as soon as the function exits, and the memory will be made available again for future function calls. There is a single stack for each *thread* of execution in a C program (until we cover multi-threading, we are dealing with a single thread).

Arrays are allocated *statically*, meaning the size of an array is specified at declaration (compile) time. The size of an array cannot be modified after it has been declared. We will see later how to *dynamically* allocate memory.

In eC, arrays share a special connection with enumerations. We already mentioned it was possible to obtain the size of an enumeration. It could be used to specify the size of an array as such:

```
double planetsRadii[Planet::enumSize];
```

However, because eC enumerations are context sensitive, in the case of array indices which always have an integer expected type, the enumeration name must awkwardly precede every value identifier:

```
planetsRadii[Planet::jupiter] = 71492;
```

To solve this issue, eC supports associating a particular array directly with an enumeration, both setting the size and the expected data type of the index appropriately for the enumeration. The above can thus be rewritten as:

```
double planetsRadii[Planet];
planetsRadii[jupiter] = 71492;
```

In this case, the array will contain 8 elements (0 through 7, or *mercury* through *neptune*).

Arrays are a very good introduction to pointers, for the simple reason that array variables are in fact *pointers*. Indeed, an array variable *points* to the start of the memory block occupied by the array elements. The only particularity which differentiates them from pointers declared as such is that array variables always point to the same array and cannot be reassigned to point to somewhere else.

Using the identifier of an array variable without the indexing operators, we are accessing the pointer to the array, which is in fact itself a variable containing the address (in number of bytes from the beginning of system memory) where the array's memory block starts. The size of pointer variables is dependent on the system architecture, and could for example be 32 or 64 bits. If we try printing the value of array without indexing, we can see where the array is allocated in memory (we're printing it as an hexadecimal address):

```
int array[10] = { 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 };
printf("array is allocated at address %X\n", array);
```

We've already briefly mention that the * symbol is used to declare pointers while introducing character strings. Note that we can assign a pointer to whichever address we please. A pointer pointing to a 0 address is said to be a null pointer. The null identifier is available in eC and should be used to initialize a pointer to a null address:

```
int * pointer = null;
```

Now let's declare a pointer variable pointing to the array:

```
int * pointer = array;
```

Both array and pointer now point to the same block of ten integers, which we have initialized with the above series. The index operator we've seen is one way of *dereferencing*, or obtaining the value at a specified address contained in a pointer or array variable. The same indexing syntax is available with pointer types as well:

```
printf("The fifth element is %d\n", pointer[4]);
```

Another way of dereferencing exists, more typical to pointer. It uses the * dereferencing operator and accesses the value of the pointer's data type at the memory location exactly where the address contained in the pointer variable points:

```
printf("The value at pointer is %d\n", *pointer);
```

It is in all ways equivalent to using the 0 index as in `pointer[0]`, and is equally available to pointer and array variables. Both syntaxes can also be used to access elements beyond the first, through pointer arithmetics. Pointer types support some arithmetic operations (with restrictions). An important concept to remember is that any operation on a pointer is performed according to its data type. Therefore an addition of 1 to a pointer does not increment the pointer by a single byte, but by the size of its data type. Therefore, the following are equivalent:

```
printf("The fifth element is %d\n", pointer[4]);
printf("The fifth element is %d\n", *(pointer + 4));
```

We can further demonstrate this by printing the addresses:

```
printf("Address in pointer is %X\n", pointer);
printf("Address in pointer + 4 is %X\n", pointer+4);
```

A sample run produced the following output:

```
Address in pointer is 402000
Address in pointer + 4 is 402010
```

As you can see, the second address is higher by 16 bytes (10 in hexadecimal), our 4 times the size of the `int` data type (4 bytes), which the pointer is declared to point to. Pointers can also be declared without specifying what data type they are pointing to, by specifying a `void` data type, like such:

```
void * pointer;
```

`void` pointers as they are commonly called do not support any arithmetic operations and must first be casted to pointers to a specific data type before performing any arithmetic, indexing or dereferencing operation with them:

```
void * pointer;
pointer + 4;          // invalid
(byte *)pointer + 4; // points 4 bytes further
```

The opposite of the dereferencing operator is the *referencing* operator, denoted by the symbol &. It returns the address of its argument. It can be used to obtain the address of any l-value expression, such as a variable, and the resulting expression is a pointer to the memory where that l-value is stored:

```
int a;
int * pointer;
pointer = &a; // pointer now points to the a variable
*pointer = 4; // variable a now has value 4
```

It is also possible to obtain the address of a specific element of an array. For example:

```
int array[10] = { 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 };
int * pointer = &array[2];
printf("The third element is %d\n", *pointer);
```

Note that the following are two completely equivalent ways of obtaining the address of the first element of an array:

```
int * pointer = array;
int * pointer = &array[0];
```

Pointers can also be used to receive a single dimension array of an arbitrary size as a parameter:

```
void PrintArray(int * array)
```

Also, observe how the following are equivalent:

```
int a;
a = 10;     // Assign 10 to a
*(&a) = 10; // Stores 10 at the address of a
```

A common usage of pointers is to pass variables by reference to a function. This allows a function to return values in another way than its return type. It also allows for both reading from and writing to the function's arguments. We could write a function taking two arguments, adding the second number to the first one as an example. The argument to be modified will be passed by reference, we will thus declare it as a pointer:

```
void AddTo(int * a, int b)
{
   *a += b;
}
```

A sample usage would be:

```
int number1 = 10;
AddTo(&number1, 5);
```

Following that call, *number1* will hold the value 15. Note that we could not write:

```
AddTo(&5, number1);
```

That is because 5 is not an l-value, it doesn't have any address and thus cannot be assigned a value; it cannot be passed by reference. Note that the precedence of * is lower than that of the . member access operator and the indexing [ ] operators. When using pointers with structures, this necessitates a somewhat cumbersome syntax: `(*pointer).member`. A special notation exists to facilitate dealing with structures: `pointer->member`. It is heavily used in both C as well as in C++, in which it has a very important connotation with pointers. We could rewrite our earlier example above as:

```
struct InventoryItem
{
   int code;
   float price;
};
InventoryItem items[3] =
{
   { 1234, 45.0f },
   { 5678, 55.0f },
   { 9012, 60.0f }
};
void PrintItem(InventoryItem * item)
{
   printf("Code: %d, Price: %.2f\n",
      item->code, item->price);
}
```

We can then call *PrintItem* with a reference to a specific element of our *InventoryItem* array writing:

```
PrintItem(&items[1]);
```

However, eC makes it possible to pass structures by reference without explicitly using pointers, as we saw earlier in the chapter about structures. The -> operator in eC is thus rarely needed in practice. The above function and function call could be written as:

```
void PrintItem(InventoryItem item)
{
   printf("Code: %d, Price: %.2f\n",
      item.code, item.price);
}

PrintItem(items[1]);
```

It might be useful to know that item is in fact a pointer, and an address could be passed to the *PrintItem* function. We can also add a check to *PrintItem* to check the non-null status of item such as:

```
if(item != null)
```

Note that this particular case (structures automatically passed by reference) is the only exception when the `!= null` is required. Pointers declared as such do not require the `!= null` and the Ecere standards discourages including it, preferring the shorter syntax:

```
if(item)
```

So far, we've always dealt with memory allocated *statically* in variables or arrays by the programming language, either on the stack or at the global level. We've seen how we can assign that preallocated memory to pointers. A crucial concept to remember when doing so is that although we are free to assign whichever address to a pointer, the responsibility of ensuring that the memory it points to is valid lies entirely with the programmer. This is why using pointers is an advanced programming topic associated with a certain level of complexity. For example, if we make a pointer declared globally point to a local variable inside a function, when that function exits, the pointer no longer point to that variable which does not exist anymore. Writing to or reading from that pointer write or read to somewhere else in memory and cause undesired behaviors. The following example illustrates this unwanted situation:

```
int * pointer;
void Function1()
{
   int a;
   pointer = &a; // Make pointer point to local a
}
void Function2()
{
   Function1();
   // Function1 exited, a no longer exists
   *pointer = 10; // Where does 10 go?
}
```

It is also possible to allocate memory *dynamically*, and assign a whole new block of memory to a pointer. However great care must be used with dynamic memory, since it does not have a defined scope. Allocated memory must thus be freed manually when it is no longer required. Not doing so will result in a *memory leak*. Memory leaks are a major problem if they are of a large size or within a recurring operation, as allocated memory is no longer available for allocation. They may eventually result in the system running out of memory. The pool of memory for dynamic allocation is called the *heap*. Large chunks of memory are ideally allocated on the heap. Fortunately, eC provides some mechanisms to ease keeping track of dynamic memory allocated through its object oriented constructs. For now, we will cover how to allocate and free memory of any basic or structure data type. Every allocation should be matched with an eventual deallocation.

The new eC operator can be used to allocate dynamic memory. It is followed by a data type and the number of elements of that data type (enclosed within square brackets) to allocate. Let's first try to allocate a dynamic array of ten integers just like our initial statically allocated array:

```
int * pointer = new int[10];
```

We can then use the pointer for read and write operations. Let's store some values in it:

```
pointer[0] = 1;
pointer[1] = 1;
pointer[2] = 3;
```

As you can see dynamic memory allocation does not support list initializers. Alternatively, the new0 operator exists which will initialize all allocated memory to 0. We could facilitate our initialization task here by storing the initialization data in a static array and then copying it over to the dynamic array. A C function exists to perform memory copy operations: *memcpy*. Its prototype is as follows:

```
void * memcpy(void * destination, void * source, uint size)
```

We could then fill in our dynamic array like this:

```
int array[10] = { 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 };
int * pointer = new int[10];
memcpy(pointer, array, sizeof(array));
```

Another C memory manipulation function allows for safely moving a block of memory when the source and destination can potentially overlap. It is simply mentioned here with its (very similar) prototype:

```
void * memmove(void *destination, void *source, uint size)
```

A last C memory function we will mention is *memset*, which allows to fill every byte of the specified memory area to a single byte value:

```
void memset(void * area, byte value, uint count)
```

It could be used to clear the memory to 0 after it has been allocated, rather than using the new0 operator as such:

```
int * pointer = new int[10];
memset(pointer, 0, sizeof(int) * 10);
```

Note that the use of a `sizeof` with a pointer is the size of the pointer variable, not of the memory it points to (which can not be known by the compiler, since pointers are dynamic). This is why we explicitly specify the size of ten integers rather than writing `sizeof(pointer)`.

When the memory is no longer needed, it must be freed with the `delete` operator by simply writing:

```
delete pointer;
```

The `delete` operator will also clear the value of *`pointer`* so that it becomes a null pointer (it will be set to a value of 0). A last memory operator exists which allows for resizing previously allocating memory without losing the data fitting within both the range of both the previous and new memory block. Memory *reallocation* is performed with `renew`. Reallocation can result in either the memory block to remain at the same address, or to be allocated at a completely separate location. The following example demonstrate a usage of the `renew` operator:

```
int array[10] = { 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 };
// First allocate only enough room for 5 numbers
int * pointer = new int[5];
// Fill in first 5 numbers
memcpy(pointer, array, 5 * sizeof(int));

// Now we want to expand to hold 10 integers
// pointer is being assigned the new memory address
pointer = renew pointer int[10];

// The first 5 numbers are still properly initialized
// Fill in the last 5 numbers
memcpy(pointer + 5, array + 5, 5 * sizeof(int));

delete pointer;
```

A `renew0` counterpart to `renew` exists which will clear to 0 only the new parts of a memory block.

It is naturally possible to have more than one level of pointers. Multidimensional arrays can be implemented as such. Note however that such an implementation does not store the entire array as contiguous memory, but rather as an array of pointers, each pointing to a separate array of values (or pointers if more than 2 dimensions). These constructs are not compatible with statically declared arrays and should not be mixed together. We will conclude this section with an example of such an array using multiple levels of indirections.

```
int ** AllocateArray(int nRows, int nColumns)
{
   int j;
   int ** array = new int *[nRows];
   for(j = 0; j < nRows; j++)
      array[j] = new0 int[nColumns];
   return array;
}
void FreeArray(int ** array, int nRows)
{
   int j;
   for(j = 0; j < nRows; j++)
      delete array[j];
   delete array;
}
void PrintArray(int **array, int nRows, int nColumns)
{
   int j; // row counter
   for(j = 0; j < nRows; j++)
   {
      int i; // column counter
      for(i = 0; i < nColumns; i++)
         printf("%d  ", array[j][i]);
      printf("\n");
   }
}
class ArrayApp : Application
{
   void Main()
   {
      int ** array = AllocateArray(3, 2);
      array[0][0] = 1;
      array[0][1] = 1;
      array[1][0] = 2;
      array[1][1] = 3;
      PrintArray(array, 3, 2);
      FreeArray(array, 3);
   }
}
```

Notice how the same indexing syntax as with static array can be used to access the array elements, but the allocation and deallocation is much more complicated. Also observe that the number of rows must be known to individually deallocate each row of the array.

Further in the book, we will learn about using object oriented classes included with the Ecere SDK to facilitate the use of dynamic arrays.

Just like data is stored in variables at a specific address, instructions making up functions are also stored in memory, and thus functions have addresses as well. Storing the addresses of functions is very useful for calling an arbitrary function from a single call in the code. One very common application in procedural programming is callback functions, whereas a function can be passed across an API as an argument and called back in the middle of another process to let the API user perform any work he desires. C has a particularly convoluted syntax for defining pointers to functions. We will nonetheless try to make some sense out of it, but be relieved that with the object oriented paradigm, callback functions are usually replaced with virtual methods and there should be very few requirements for declaring function pointers in eC. Consider this very simple function:

```
int AddTwoNumbers(int a, int b)
{
   return a + b;
}
```

We can declare a pointer to such a function as such:

```
int (*addTwoNumbersPtr)(int a,int b) = AddTwoNumbers;
```

Note how the identifier for the pointer being declared (*addTwoNumbersPtr*) figures where the function name would normally be, preceded by the familiar asterisk indicating we're declaring a pointer and enclosed in parentheses. Although an initializer was used inside the declaration, it is optional and the pointer could be assigned a value later on. Calling a function pointer has the exact same syntax as a call to a regular function:

```
addTwoNumbersPtr(10, 20);
```

Because of the confusion often associated with such pointer declarations, it is commonplace to define a `typedef` for function pointers. The actual pointer declaration then looks much more familiar. Using a `typedef` to achieve the same as above would look like this:

```
typedef int (*AddTwoNumbersPtr)(int a, int b);
AddTwoNumbersPtr addTwoNumbersPtr = AddTwoNumbers;
```

Finally, a cast operation includes no identifier following the asterisk. The following example calls the function *AddTwoNumbers* by casting a `void` pointer containing its address:

```
void * pointer = AddTwoNumbers;
((int (*)(int a, int b))pointer)(10, 20);
// Same as: (using typedef)
((AddTwoNumbersPtr)pointer)(10, 20);
```

# Section 2

## Object Oriented Programming with eC

Classes, methods and instances

Inheritance

Polymorphism and virtual methods

Properties

Encapsulation and access control

Importing and working with multiple modules

Name Spaces

Units and conversion properties

Bit collection classes

***Classes, methods and instances***

By emptiness of self-aspect or self-character,
therefore, is meant that each particular object
has no permanent and irreducible
characteristics to be known as its own.

*-- D.T. Suzuki*

Designing software in terms of objects can greatly improve the maintainability and expandability of applications. Furthermore, if properly used, it can lay out an efficient development plan and speed up the entire engineering process.

In the previous section, we've learned how to organize blocks of data into structured data types. We have also covered how one occurrence of any data type is an *instance* of that data type and seen how to use eC's instantiation with structure data types.

We will now learn how to bring our objects to life by endowing them with *methods*. We will also learn how eC considers various kinds of data types to be *classes* (and to a certain extent, all data types), the structures defined with the `struct` keyword being only one kind among others.

A *method* describes how to perform a certain action with any instance (object) of a specific class. Therefore a method is implemented as a function belonging to a specific class. Methods have an implied parameter (called the `this` object) representing the object on which the method was called.

The `this` object can be referenced explicitly through the `this` keyword, but it can be omitted when accessing members of the class (either data members, methods or properties) because all identifiers are first matched against them, as if prefixed by "`this.`". However, if a local variable or function parameter of the same name exists, it becomes necessary to explicitly write the "`this.`" to differentiate members from local definitions (the latter having precedence). Here is a rewrite of an earlier structure example using methods:

```
struct InventoryItem
{
   int code;
   float price;

   void Print()
   {
      printf("Code: %d, Price: %.2f\n", code, price);
   }
};
```

Notice how we no longer require *Print* to take a parameter, as the item to be printed is the this object. We also renamed our function *PrintItem* from last section to simply *Print*, as Item is implied from the class to which the method belongs. We can access the code and price data members of the InventoryItem directly, but they could also be accessed as "this.code" and "this.price". Method definitions can figure within a structure just like data member declarations. Observe how the Print function and it's body's compound block are indented an additional 3 spaces to the right compared to a regular function, to be in line with the data members declarations. Let's now look at an example of how to call that *Print* method:

```
InventoryItem item { 1234, 45.0f };
item.Print();
```

This declared an instance of the *InventoryItem* class to be referred to by the identifier *item* through the instantiation syntax[12]. The Print method is then called on item. Methods, just like structure data members, are members and are thus accessed through the '.' member access operator. Following precedence and associativity rules, *item.Print* is the expression being called through the () call operator. Just like with data members, the "*this*." is not necessary when calling another method of the same class from within a class method. Method calls are equally valid on anonymous instantiations, so that the following would work:

```
InventoryItem { 1234, 45.0f }.Print();
```

Our use of structures so far has been confined to their scope, as they are statically allocated. It is possible to allocate them dynamically, as we've seen the new operator accepts any data type. However, doing so requires using pointers and their related complexity and cumbersome syntax:

```
InventoryItem * item = new InventoryItem[1];
item->code = 1234;
item->price = 45.0f;
```

Therefore, eC introduces an easier to manage kind of classes which does not present these caveats. Structures will typically be allocated dynamically only in an array type scenario, when numerous elements must be allocated in contiguous memory. Indeed, memory of other kinds of classes can not be allocated as contiguous arrays. Additionally, structure possess other advantages over the classes we will describe next, especially for small, self-contained data type which do not require a construction or destruction process. One of these advantages is their compatibility with standard C structures.

12 See the chapter on structures in the first section for a full description of the instantiation syntax.

The more powerful classes we shall look into now are defined using the `class` keyword typical of other object oriented languages such as C++. It is also the same kind of class we used to define our main application class in our very first Hello, World program. Instances of such classes are *always* allocated dynamically. This differs from C++ where objects can be declared statically (automatic variables, using C++ terminology: `MyClass object;`) or dynamically (through the C++ `new` operator: `MyClass * object = new MyClass;`). The eC `new` operator is not needed to allocate these classes: instead, the usual instantiation syntax is used. We could implement our *`InventoryItem`* in a very similar manner:

```
class InventoryItem
{
public:
   int code;
   float price;

   void Print()
   {
      printf("Code: %d, Price: %.2f\n", code, price);
   }
}
```

And an instantiation would be equally simple:

```
InventoryItem item { 1234, 45.0f };
item.Print();
```

An important point to underline here is that classes defined with the `class` keyword (from now on referred to as *regular* classes) do not support declarations following their definition blocks as `struct` classes do. For that reason, the semicolon following the class definition block is not required (but still allowed). However, for `struct` definitions, it is mandatory and forgetting it will result in severe (sometimes cryptic) syntax errors.

As you can see our definition of *`InventoryItem`* as a regular class is very similar to the previous one as a structure. There is however important subtle differences. First you might have noticed the necessity for the `public` keyword. We will discuss member access control and the differences between private and public members in a dedicated chapter; let's just mention for now that by default, structure members are public, whereas regular class members are private unless otherwise specified. As we said, regular classes are allocated dynamically. We mentioned in the chapter on pointers and memory that this signifies we must make sure the memory dynamically allocated is freed when it is no longer required. Furthermore, regular classes undergo construction and destruction processes.

Fortunately, eC greatly facilitates the process of managing the destruction of these instances. Depending on where they are declared, instances may be automatically destructed. The global instances for example, are automatically destroyed on the program's termination. Also, objects which are data member of other classes automatically get freed up when instances of that class are destructed.

Other object instantiations must be freed manually with the `delete` operator. The following example illustrates these various scenarios. It uses out last *InventoryItem* class definition.

```
InventoryItem globalItem { 1234, 45.0f };

class InventoryApp : Application
{
   InventoryItem memberItem { 5678, 50.0f };

   void Main()
   {
      InventoryItem localItem { 9012, 55.0f };

      globalItem.Print();
      memberItem.Print();
      localItem.Print();

      delete localItem;
   }
}
```

Notice how only *localItem* needs to be freed manually. The *globalItem* will be destroyed automatically when the program terminates, and *memberItem* as well when the *InventoryApp* instance is destructed. Regular classes may be instantiated anonymously, but the resulting expression must be stored somehow so that it can be freed at some point. Otherwise it will result in a memory leak. A handle for holding on to an object is declared like a regular variable:

```
InventoryItem item;
```

This does not instantiate or allocate any memory for an *InventoryItem* object. This is not the same as `struct` classes, for which the only difference between the above syntax and an instantiation syntax is that the above is uninitialized, whereas using the instantiation syntax everything is initialized to 0. Once declared, a handle can then be assigned an anonymous instantiation as such:

```
item = InventoryItem { 1234, 45.0f };
```

The instantiation could also be done within the initializer:

```
InventoryItem item = InventoryItem { 1234, 45.0f }.
```

Since *InventoryItem* is expected here, this could be written simply as:

```
InventoryItem item = { 1234, 45.0f };
```

Note that for readability purposes, omitting the class name should only be done when the expected data type is obvious at the location of the instantiation, so that we know what class is being instantiated. It is important to point that *global* and *member handles* are *not* automatically freed, since they are not by themselves instantiations. However, it is possible to use the variable of an instantiation as a handle, and thus reassign a new value to an instance (ensuring the previous will also be freed). This should be done with care. One thing to keep in mind is that if it was declared as a global or member instantiation, such an instance *will* be automatically freed, even if it was assigned a new value either from an anonymous instantiation or from another named instantiation.

This automatic management of global and member instances is done through *reference counting*. Instances of regular classes contain a *header* featuring among other things a reference count, indicating whether an object is still referenced by different parts of the program. Every time an instance of a regular class is deleted with the `delete` operator, its reference count is decremented by one. When it reaches zero (or goes negative), the object is destroyed. Reference counting in eC is only done for these two specific cases, as well as on a manual basis. This differs from other programming languages where the entire garbage collection process relies on automatic reference counting, which can have a significant performance impact. Because reference counting in eC is only done in key places (on which the developer has great control), it does not affect the application performance. We will cover the use of reference counting in details in the last section of this book on *Advanced eC programming*.

Upon instantiation, the reference counter of an object is set to zero by default. Thus it will be deleted upon the first use of the `delete` operator, as it reaches -1 (since it is smaller or equal to zero). As we start using them, you will find that many built in classes in the Ecere SDK can actually manage themselves: their associated system will handle their deletion. Example of this are the *Window* and derived classes which we will use for building graphical user interfaces, and the *Socket* class which we will use for network communication. These will often not require the use of the `delete` operator, even when instantiated anonymously or as a local variables.

We started talking about construction and destruction. With regular classes, the construction process involves the allocation of memory and the construction of any instance members. The destruction process naturally includes the destruction of all member instances. Additionally, it is possible to define default values for members by including assignments directly at the class level, as such:

```
class InventoryItem
{
public:
   int code;
   float price;

   code = 1234;   // Default value for code
   price = 45.0f; // Default value for price
}
```

Arbitrary code can also be executed by defining a constructor and or a destructor, which are respectively called while constructing and destructing the instance. The constructor is simply defined as a member method named after the class, whereas the destructor is named with the class name preceded by a tilde (~) symbol (following the C++ syntax):

```
class InventoryItem
{
public:
   int code;
   float price;

   InventoryItem()
   {
      printf("Constructing InventoryItem object\n");
   }
   ~InventoryItem()
   {
      printf("Destructing InventoryItem object\n");
   }
}
```

However, constructors particularly do not play a role a important as in C++, for example. Neither constructors nor destructors can take in any parameters, and only a single one of each can be defined within a class. Instead, members can be directly assigned a value through the instantiation syntax initializers (either through the data members, or the properties which we will describe in next chapter). They cannot be specified a return type either. A constructor should never fail, but returning `false` (they have an implicit `bool` return type) will result in a the object instantiated to be null.

At the beginning of this chapter we've brought up the concept of a this object, which the methods are dealing with. Normally, these are assumed to be of the class data type. In eC, it is possible to override the data type for the this object. This is useful for some scenarios, one of which is when we want to regroup within a class methods which do not require dealing with a particular instance. We can specify the data type by preceding the method's name by a data type, followed by a double colon. Let's look at an example of defining a method which does not require an instance to be called (a *static* method in C++, although the static keyword never bears that meaning in eC). We describe such a method by simply not writing anything immediately before the double colon (do not get confused with the separate void return data type):

```
class OtherClass
{
   void ::DoSomething()
   {
      // There is no this object available here
   }
}
```

We can then call the *DoSomething* method either the usual way, as an *OtherClass* instance member, or by invoking the method directly from the class using the double colon operator:

```
OtherClass::DoSomething();
```

Similarly, we can define the this object to be of a specific data type. These features are also available for structures. That method could expect an instance of our earlier *InventoryItem* class:

```
class OtherClass
{
   void InventoryItem::PrintItem()
   {
      Print();
   }
}
OtherClass object { };
InventoryItem item { };
```

Notice how we can directly call the *Print* method of the *InventoryItem* class, because our this object is an *InventoryItem*. Considering the above two instantiations, these are two valid ways to invoke the *PrintItem* method with *item* becoming the this object:

```
object.PrintItem(item);
OtherClass::PrintItem(item);
```

When explicitly specifying as the method's this object data type the class it is declared in, it indicates that the method may deal with a different instance of the class than the object it was called from. An example of such a method is given here:

```
class OtherClass
{
   void OtherClass::DoSomething()
   {
      // this object here is not the calling instance

   }
}
OtherClass object1 { };
OtherClass object2 { };
```

The *DoSomething* method must be called with an *OtherClass* argument, even if called from an *OtherClass* instance:

```
object1.DoSomething(object2);
OtherClass::DoSomething(object2);
```

It is important to note that the data type for the method will not change in derived classes (we will learn about deriving classes in the chapter on inheritance). Furthermore, this particular type of method is known as an *adaptable method* and we will learn more about it in the related chapter, also discussing instances virtual methods. Their use will be common for the notification events used by graphical user interface elements to communicate between them. However, it is not important to fully understand these details of their inner workings at this point.

It is also allowed to specify a this object type for regular functions, outside of any class, which will result in the this object being accessible as if the function was a regular method part of that class:

```
void InventoryItem::PrintItem()
{
   // Call InventoryItem's Print method as if we were
   // inside a class method
   Print();
}
```

As of the time of writing this book, eC only supports a single instantiation per declaration. Thus the following is not valid:

```
InventoryItem item1 { }, item2 { };
```

We already stated that it is not possible to allocate regular classes in contiguous arrays like it is for `struct` classes. However an array of handles can be used, each element being instantiated and freed individually:

```
InventoryItem items[10];
int c;

// Allocate items
for(c = 0; c < 10; c++)
   items[c] = InventoryItem { };

...

// Free items
for(c = 0; c < 10; c++)
   delete items[c];
```

***Inheritance***

All under Heaven have a common Beginning
This Beginning is the Mother of the world.

*-- Tao Te Ching, verse 52*

We already learned how to create hierarchies by defining data members of a class using another class as a data type. We will now explore how to expand classes through another dimension: inheritance. An inherited class contains all the members of the *base class*, plus additional members specific to the *derived (inherited) class*. An important concept is that an instance of the derived class is a valid instance of the base class, and as such a cast to the base class is perfectly valid. An instance of a derived class can also be used where an instance of the base class is expected, without an explicit cast.

In eC, most kinds of data types support inheritance. In fact, we will learn that some types can even inherit off a different kind of data type. The base class is specified after a colon in the type definition. We already encountered the inheritance syntax a few times in the previous chapters, remember the following:

```
enum Element : byte { metal, wood, water, fire, earth };

enum PlanetOrDwarfPlanet : Planet
```

In the first example, the *Element* enumeration is to be derived from the byte system data type, whereas the *PlanetOrDwarfPlanet* enumeration is to be derived off the *Planet* enumeration. Remember also the special entry point *Application* class:

```
class HelloApp : Application
```

This actually defines a new class *HelloApp*, inheriting from the *Application* base class. Let's work with an example where we will create both a base and derived class:

```
class BaseClass
{
   int a;
};

class DerivedClass : BaseClass
{
   int b;
};

DerivedClass test { a = 10, b = 20 };
```

Notice how the member *a* from *BaseClass* is available in *DerivedClass*. Now let's demonstrate that the *test* instance is indeed a valid *BaseClass*:

```
void PrintBase(BaseClass object)
{
   printf("a = %d\n", object.a);
}

class InheritApp : Application
{
   void Main()
   {
      PrintBase(test);
   }
}
```

You can see how we called the global function *PrintBase* with a *DerivedClass* instance (*test*), when it expects a *BaseClass*. *PrintBase* can successfully print the member *a* of *DerivedClass*, because it inherits from *BaseClass*. However, using a *BaseClass* instance where a *DerivedClass* instance is expected will generate a compiler warning. Consider the following code:

```
void PrintDerived(DerivedClass object)
{
   PrintBase(object);
   printf("b = %d\n", object.b);
}

class InheritApp : Application
{
   void Main()
   {
      BaseClass base = test; // No need for a cast
      // Generates a warning without the cast:
      PrintDerived((DerivedClass)base);
   }
}
```

An explicit cast must be performed to eliminate the warning. Such a cast should only be done if we are certain the object we are dealing with is indeed a *DerivedClass* instance.[13]

13 Regular classes contain runtime type information (RTTI) identifying which class they are an instance of, but we will only cover this topic in the advanced eC programming section since it should be used with great care. RTTI is often abused and therefore it is encouraged not to rely on it to identify objects, and rather reorganize the code in such a way that it is not required. Polymorphism (the subject of next chapter) is often a solution.

A class not only inherits data members, but methods and properties as well. We will rewrite the above global functions using methods instead.

```
class BaseClass
{
   int a;

   void Print()
   {
      printf("a = %d\n", a);
   }
};

class DerivedClass : BaseClass
{
   int b;

   void Print()
   {
      BaseClass::Print();
      printf("b = %d\n", b);
   }
};

DerivedClass test { a = 10, b = 20 };

class InheritApp : Application
{
   void Main()
   {
      BaseClass base = test;
      // This will print both Derived and Base
      test.Print();
      // This will print only Base
      base.Print();
      // This will print both Derived and Base
      ((DerivedClass)base).Print();
   }
}
```

Note how both the base and derived classes define a *Print* method, each printing the member specifically defined in the respective classes. Just like our global *PrintDerived* function invoked *PrintBase* to print the base member, the *DerivedClass Print* method invokes the *BaseClass Print* method. We must specify we intend to call the *Print* method of the *BaseClass* using the double colon operator (::) to do so, otherwise the *DerivedClass Print* method will call itself recursively.

As you can see in the various call examples within the *Main* method, when methods with the same name are defined at multiple inheritance levels of a derived class, the data type of the instance on which it is called always determines which method is called, unless the method is virtual (the next chapter will explain virtual methods). The highest level method of the instance data type is the one being called.

The action of redefining a member with the same name as a base member is called *overriding*, and can be done with any kind of members (data, property or method). We will see that overriding is most often done with virtual methods, which enables calling derived methods from an instance declared as the base class (e.g. enabling the second call example to print both Derived and Base without a cast). Any other overriding should be done with great care, with proper attention paid to the member access control rules.

In eC, by default inheritance is *public*, meaning all inherited members from a base class retain their same access rights. It is possible to inherit privately, therefore making all inherited members *private* by preceding the base class by the `private` keyword. The `public` keyword can also be used to explicitly mark the inheritance as public. The particularities of public versus private members will be covered in details in the upcoming chapter on access control.

Each level of inheritance of a class can have its own pair of constructor and destructor, which will all be executed in sequence. The base class is constructed first and destructed last.

A derived class can have its own member initializers for data members and properties, and can also initialize the members of a base class. For example *DerivedClass* could have a default value for the *a* member defined in *BaseClass*, which will override any default value set at the *BaseClass* level.

```
class DerivedClass : BaseClass
{
   int b;
   a = 10;
}
```

Inheritance can also be performed with structures, with a few things to keep in mind. Data members cannot be overridden in derived classes. Remember also that structures do not support either constructors, destructors, nor class members initializers.

It is also possible for a regular class to inherit off a structure, yielding some kind of hybrid between the two kinds of data types. Such a class can have data initializers, constructors and destructors, but does not contain any extra header information like a regular class.

As such, its memory is directly compatible with a structure type and does not take up any extra space, but it does not support reference counting, run-time type information and does not have a virtual method table. Instantiation of the derived class however works exactly like a regular class: it is always allocated dynamically. Consider the following example:

```
struct BaseStruct
{
   int a;
};

class DerivedClass : BaseStruct
{
   int b;
   a = 10;
}

class InheritApp : Application
{
   void Main()
   {
      DerivedClass test { b = 20 };
      delete test;
   }
}
```

Note how the instance must be deleted, just like a regular class.

A special syntax also exists to define a class without any header information, but not inheriting from any structure:

```
class DerivedClass : struct
{
   int b;
}
```

***Polymorphism and virtual methods***

Form that includes all forms,
image without an image,
subtle, beyond all conception.
*-- Tao Te Ching, verse 14*

In the context of a programming language, polymorphism refers to being able to deal with multiple data types through a consistent interface. eC supports a kind of polymorphism through its inheritance mechanism, known as *subtyping polymorphism*. In subtyping polymorphism, instances of derived classes can be used where an instance of a base class is expected. It is further enhanced by the conversion properties which we will cover in a following chapter along with unit types. Conversion properties enable additional data types, not related by inheritance, to be usable by establishing how such conversions should be performed.

In the previous chapter, we started explaining how virtual methods enable method calls on base class instances to invoke methods defined in derived classes. This is done by first qualifying the definition of the base method with the `virtual` keyword, then overriding the method in a derived class. Let's consider our example from last chapter using a virtual method:

```
class BaseClass
{
   int a;
   virtual void Print()
   {
      printf("a = %d\n", a);
   }
};
class DerivedClass : BaseClass
{
   int b;
   void Print()
   {
      BaseClass::Print();
      printf("b = %d\n", b);
   }
};
```

As you can see, it is not necessary to use the `virtual` keyword again when overriding the *`Print`* method in *`DerivedClass`*. That is because all methods named *`Print`* in derived classes will automatically override the virtual method in *`BaseClass`*[14].

14 Unless the base virtual method is no longer accessible due to access control rules.

It is also possible not to define a body when defining a virtual function, simply by writing a semicolon (;) after the parameter parentheses, like such:

```
class BaseClass
{
   int a;
   virtual void Print();
};
```

In this case, even if not overridden, the *Print* method can still be called, but will not perform anything. If the method returns a value, it will return a default `bool` value `true` (or an integer 1).[15]

Let's now look at the call to our virtual method:

```
DerivedClass test { a = 10, b = 20 };

class InheritApp : Application
{
   void Main()
   {
      BaseClass base = test;
      // This will now print both Derived and Base
      base.Print();
   }
}
```

Even though the calling instance is declared as a *`BaseClass`*, the method overridden in *`DerivedClass`* will be called. That is because the instance *base* is pointing to (*`test`*) keeps a *virtual method table* indicating which *`Print`* method should be called. Its virtual method table is the one associated with the class used to instantiate it, *`DerivedClass`*.

It might be important to note at this point that the destructors of regular classes are always virtual in eC. Instances of classes without header information seen in last chapter (such as those inheriting off a structure) do not contain runtime class information nor a virtual table, and can therefore only rely on the data type of the instance when calling a method or deleting an object. It is thus primordial to ensure they are declared as their highest level class when using the `delete` operator on them, since the higher destructors will otherwise not be invoked.

15 Although the compiler will not enforce it, a virtual method returning a data type incompatible with a `bool`, such as a `double` or a pointer type should provide a proper base implementation.

Inside a class, a virtual method definition is in some aspects a kind of storage. The virtual method table indeed stores a pointer to the appropriate method associated in each entry. It is possible to override a virtual method through an assignment to another method:

```
class DerivedClass : BaseClass
{
   int b;

   Print = PrintMembers;

   void PrintMembers()
   {
      BaseClass::Print();
      printf("b = %d\n", b);
   }
};
```

Furthermore, the method needs not be a member of the class, but could be a compatible global function, either through its `this` type or an instance parameter. eC is very permissive in that respect for assessing the compatibility of function pointers. For example the following global definitions of *`PrintMembers`* would work equally well:

```
void DerivedClass::PrintMembers()
{
   BaseClass::Print();
   printf("b = %d\n", b);
}

void PrintMembers(DerivedClass object)
{
   object.BaseClass::Print();
   printf("b = %d\n", object.b);
}
```

In all examples up until now, instances have had their virtual method table pointing to the one defined in their respective class. eC introduces the possibility to override virtual methods at the instance level as well. For the sake of efficiency, instances only dissociate their virtual table from the virtual table of their class when a method is explicitly overridden at the instance level. The assignment syntax for overriding a method inside an instantiation is very similar to a data member assignment:

```
DerivedClass test { a = 10, b = 20, Print = PrintMembers };
```

In addition to assigning methods defined elsewhere (either global functions or class member functions), it is possible to override virtual methods by writing a new function right inside the instantiation. The syntax requires any previous initializer list to be terminated by a semicolon. We could override *Print* inside our *test* instantiation as such:

```
DerivedClass test
{
   a = 10, b = 20;

   void Print()
   {
      BaseClass::Print();
      printf("b = %d\n", b);
   }
};
```

If you manually start writing the above *Print* method definition inside an instantiation, you might notice a unique Ecere IDE feature whereas the definition is automatically completed for you as you type the opening parentheses. The return type for the function is corrected, the parameter list is filled up (according to how the base virtual function was defined), and the compound statement curly brackets are put in place, leaving the cursor in the proper position to start writing the function implementation.

Overriding virtual methods at the instance level provides additional flexibility and dynamism to eC objects. It plays an important role in defining events and customizing controls when building graphical user interfaces with the Ecere SDK.

We previously mentioned the concept of *adaptable methods*, which are defined by explicitly specifying the `this` type of a virtual method as such:

```
class BaseClass
{
   virtual void BaseClass::DoSomething()
   {

   }
}
```

It is said to be adaptable because when overriding such a method inside a member instantiation of another class derived from the specified class (*BaseClass*), the `this` object automatically adapts to the containing class. The following example illustrates the situation:

```
class DerivedClass : BaseClass
{
   int a;
   BaseClass test
   {
      void DoSomething()
      {
         // this type is adapted to a DerivedClass
         a = 10;
      }
   };
}
```

Because the *DoSomething* method has been declared as an adaptable method (by specifying it takes a different *BaseClass* object), when overriding it inside the instantiation of *test*, which is itself within the definition of *DerivedClass* (a class inheriting off *BaseClass*), the this type within the overridden method becomes *DerivedClass*.

The concept of adaptable methods has many prerequisites which might make it difficult to grasp, but it is very useful in the context of defining graphical user interfaces. Note that in common usage of the SDK, adaptable methods are used more often than they are defined, in a way that feels very natural. We will look at them again from a more practical point of view when defining events for controls inside forms. A more generic type of adaptable methods also exists which has less prerequisites, and enables the this object to be adapted to any unrelated containing class. It is defined using the any_object keyword:

```
class BaseClass
{
   virtual void any_object::DoSomething();
}

class OtherClass
{
   int a;
   BaseClass test
   {
      void DoSomething()
      {
         // this type is adapted to an OtherClass
         a = 10;
      }
   };
}
```

***Properties***

> Quantum theory requires us to give up the idea that the electron, or any other object has, by itself, any intrinsic properties at all.
>
> *-- David Bohm*

eC places a lot of emphasis on properties. Properties are sort of half-way between methods and data members. They provide methods to *set* or *get* the property of an object, which is often stored internally in a data member.

The advantages of properties are multiple. First, they allow for instant visual feedback, for example in the case of the Ecere IDE's Object Designer. Code to reflect modifications can be executed in the same operation as the assignment of a new value. The IDE has a property sheet in which you can visualize and change the values of properties, and get instant feedback in the visual designer window. The IDE's Object Designer is extremely powerful and versatile[16]. This system all relies on the properties and was one of the original reasons for designing eC and the Ecere Component Object Model.

Second, properties present this layer of protection over the data members. A property can be made public whereas the data member actually used to store the property can remain private. Properties can also only contain a `set` or a `get` method, therefore making it write-only or read-only. A property can also return or set a value which is not stored exactly as such in the class. eC has a very clean syntax for defining properties:

```
class Person
{
   int age;
public:
   property int age
   {
      get { return age; }
      set { age = value; }
   }
}
```

Notice how the definitions of `set` and `get` methods do not include any return data type or parameters list, not even the familiar empty parentheses. Of course, only one of each can be defined. The special identifier *`value`* is available within the definition of a `set` method. It represents the value to which the property is being set. A property will evaluate as the the return value of its `get` method.

16 We will cover using the object designer and property sheet in a dedicated chapter of next section.

You might also have realized we used the same identifier (*age*) for both the internal data member in which we store the age, and the actual property. This is allowed in eC, and special rules exist to differentiate whether you refer to the *age* data member or to the *age* property (although typically, the default behavior satisfies most needs). When used inside a method of the class itself, the data member has priority. When used outside, the property has priority. To enforce the use of the property from inside, it can be accessed by preceding the member with "`property::`", for example our age property could be internally accessed as: `property::age`. It is also possible to enforce the use of the data member by obtaining its address (since a property does not have an address), and then dereferencing it, as such: `*(&age)`. Member access control also plays an important role in selecting a data member versus a property. Indeed, inaccessible members can not be selected. Regular classes data members are usually kept private, whereas properties are made public.

Here is some examples of how one could set the *age* property for a *Person* object, just as if it was a data member:

```
Person person { 20 };
Person person { age = 20 };
```

We could of course include some more code within our `set` and `get` methods. Just as an example, we could print out to the console that the age property is being changed:

```
set
{
   printf("age (previously %d), is now %d", value);
   age = value;
}
```

Although functionally equivalent to separate set and get methods (for example a method `void SetAge(int age)` and another one which would go `int GetAge()`, as it is common in C++), properties behave exactly like data members. There is of course some limitations; one of these is that the referencing operator cannot be used on them, since properties do not have an address. For example, the following operations can be performed with them, as if they were regular data members:

```
person.age = 100 - person.age;
person.age += 10;
person.age *= 2;
```

These all go through the `set` and `get` methods to do their work.

Properties do not necessarily need to be stored inside a data member. As an example, we could instead decide to store the year of birth of that person and compute the age accordingly. We'll assume the person is born on January 1st, and store the current year in a global variable for the sake of simplicity:

```
int currentYear = 2008;

class Person
{
   int yearOfBirth;
public:
   property int age
   {
      get { return currentYear - yearOfBirth; }
      set { yearOfBirth = currentYear - value; }
   }
}
```

Because they can execute code upon initialization, properties can often play the role constructor parameters play in the C++ language. However, that must be done with care, as each property being set is executed one at a time. Often actions require more than one parameters to previously be set. Although repeating such action on setting any of its property is one solution, it may pose a performance problem if the process is the least computing intensive. In these cases, a separate method to perform the action can be defined, either using the values of already set properties or taking the values as parameters. Another possible solution is to set all these properties together as a structure or class object, or less ideally, only performing the action on the last property to be set.

It is important to know that properties are set in the order in which they are initialized. Furthermore, the order of unnamed initializers follow their declaration order, and they can be intertwined with data members.

Just like for data members, a class can define a default value for a property (which will execute the proper `set` method) right inside the class body.

In addition to the `set` and `get` methods, eC supports defining a method whose purpose is to indicate whether a property has been set or not. The `isset` method therefore, if defined, therefore allows an additional *unset* state for a property. It can be implemented internally through any mean, for example a simple boolean data member can maintain the *set* status. An unset state may be useful in many scenarios, for example in objects inheriting a default attribute from a multi-objects hierarchy. Here is a sample usage of `isset`:

```
int defAttrib = 10;

class MyClass
{
   bool attribSet;
   int attrib;
   property int attrib
   {
      set { attrib = value; attribSet = true; }
      get { return attribSet ? attrib : defAttrib; }
      isset { return attribSet; }
   }
}
```

eC does not support virtual properties; a property belongs to a specific class and should keep its meaning throughout the derived classes which publicly inherits it. However, eC provides a mechanism allowing a derived class to react (or adapt) to setting an inherited property of a base class. The watch keyword is used alongside the property to be monitored within the class definition to achieve this. A derived class can *watch* the property of its base class, and act on it. The following example demonstrates how it's done:

```
class BaseClass
{
   int attrib;
   property int attrib
   {
      set { attrib = value; }
      get { return attrib; }
   }
}

class DerivedClass : BaseClass
{
   void AdaptToAttribChange()
   {
      // React somehow to the change of attribute
   }

   watch(attrib)
   {
       AdaptToAttribChange();
   }
}
```

Any time the *attrib* property of the base class is set, the derived class can thus invoke the *AdaptToAttribChange()* method, giving it a chance to react to the modification.

### *Encapsulation and access control*

The Master sees things as they are,
without trying to control them.
She lets them go their own way,
and resides at the center of the circle.
*-- Tao Te Ching, verse 29*

A concept fundamental to object oriented programming is encapsulation. Encapsulation consists in hiding implementation details from an interface, to prevent changes to the implementation from impacting external pieces of code. In object oriented programming, the major part of interfaces is presented in the form of classes, and thus classes encapsulate a part of a program's functionality in a convenient interface.

Encapsulation is closely related to access control, which effectively separates the implementation internals from the interface. The approach regarding access control taken by eC is very practical, driven from the typical usage of application programming interfaces. It is considerably different from that of C++, for example. The design starts from the assumption that a limited number of developers should work on a single module at a time, each of them having a good understanding of all classes within the module they are working on. It thus encourages modularity rather than sacrificing performance for safety in the interaction between classes of a single module.

There are two types of modules in eC. The first type is simply a single eC source file (with a .ec extension), whereas the second takes the form of a dynamic shared library. We will cover in depth how to work with multiple modules through a process known as *importing* in the next chapter. The access control system of eC offers three distinct modes dictating which kind of module will be allowed to access specific code constructs.

Global definitions (structures, classes, enumerations, functions, ...) can have an access control specifier, and member declarations within a class or structure support access control as well. Let's first look at how each of the three modes affect global definitions. The global access mode is *private* by default. Global definitions declared as private are accessible throughout the same target module (a single *project* in the Ecere IDE, either a shared library module, or the main application module executable). Private definitions are not accessible outside the target module. To make them available outside, they must be made *public*. They can be declared as public individually by preceding each one with the `public` keyword, or the current global access mode can be changed by writing `public:` in the code at the global level. Similarly, the global access mode can be reverted back to private by writing `private:`. The following code illustrates the syntax:

```
	// Function1 is private (default global access mode)
	void Function1()
	{
	}

	// Class1 is private (default global access mode)
	class Class1
	{
	}

	// Class2 is specifically marked as public
	public class Class2
	{
	}

public:
	// Function2 is public (global access is now public)
	void Function2()
	{
	}

	// Class3 is public (global access is now public)
	class Class3
	{
	}

	// Class4 is specifically marked as private
	private class Class4
	{
	}

private:

	// Function3 is private (global access private again)
	void Function3()
	{
	}
```

The remaining access mode is *static*, which takes on the same meaning as the `static` keyword in C for qualifying global definitions. Static definitions are only available within the same source file module (.ec). In eC, `static` can be used the same way as public and private followed by a colon (`:`) to change the default global access mode, and can qualify all kinds of definitions (whereas it can only qualify variable definitions in standard C). Non static (both public and private) constructs, including variables, are automatically available in other eC source files, whereas C requires the use of the `extern` keyword to declare them as existing in another module.

While we're covering the static keyword, let's mention the only other usage for it in both C and eC. When used to qualify a variable inside a function, that variable will subsist throughout the entire program's life and always refer to the same memory. It will not be allocated through the stack as a regular local variable would. It is thus somehow equivalent to a global variable, but still limited in scope to the compound statement within which it is declared. As such, particular care must be given to static local variables when dealing with multiple threads, as they are not inherently thread-safe the way variables allocated on the stack are (each thread having its own stack). Static local variables are useful for keeping track of values between subsequent calls to a function, or to hold large chunks of memory not suitable for the stack, but which we still want to localize inside the particular function where it is used. Again, it is important to ensure the function is not called from two different threads at the same time. The following example demonstrates a static local variable:

```
void TestFunction()
{
   // counter will only be initialize to 0 on the
   // first call to TestFunction()
   static int counter = 0;
   counter++;
   printf("TestFunction has been called %d times\n",
      counter);
}
```

As a reminder, the static keyword is not used with any other different meaning in eC (except along the import keyword), unlike C++ which uses it for defining methods not working on a specific instance of a class. Instead, that is done by specifying the method does not have a this type (see previous discussion in the chapter on *Classes, methods and instances*).

It is important to note that to export a construct using public or private requires all dependencies for that construct to be accessible at the same level. The exposed interface of public constructs cannot use private or static constructs, whereas private constructs cannot make use of static constructs. For example, a public function cannot use a private or static class for its return type or any of its parameters. Similarly, a private function cannot make use of a static class. Doing so will generate a compiler error. Here is a sample conflicting scenario:

```
class PrivateClass { }
public void PublicFunction(PrivateClass object) { }
```

This results in the following error from the compiler:

```
error: Public function making use of a private class
```

Also, public classes can only inherit from private classes if the inheritance is private (see previous chapter on inheritance discussing the difference between private and public inheritance). Currently, static classes can only be inherited from classes which are static as well. For example the following reports an error (inheritance is public by default):

```
class PrivateClass { }
public class PublicClass : PrivateClass { }
```

Whereas it is fine if inheritance is done privately:

```
class PrivateClass { }
public class PublicClass : private PrivateClass { }
```

Finally, a special global declaration access mode exists for these times when absolute, utter, complete, unequivocal compatibility with C is required. It is specified in exactly the same way, using the `default` keyword. In that compatibility mode, nothing is exported through the eC mechanisms. Non static variables declared in `default` mode must thus be accessed from other source files by declaring them using the `extern` keyword. Structures and enumerations declared with `default` only support the standard C usage (used through a `typedef` or by preceding the structure identifier with the `struct` keyword). As such, they will not be available in the component object model system (for features such as reflective programming, live documentation, dynamic class constructions). Classes, which are simply not available in standard C, will not be affected by the `default` mode, since they do not introduce any potential conflict. They should however not be declared using this mode to prevent any ambiguity (they would fall back to a private access mode). Another implication of the `default` definition mode is that it will prevent any mangling from name spaces (effectively putting the definition in the global namespace). When including header files[17] ending with the *.h* extension, the global definition mode is automatically set to `default`. eC headers should end with a *.eh* extension to have the usual private access mode.

Both public and private access control modes also apply to qualifying members of structures and classes, enabling encapsulation of what is to be kept private inside a specific module. As of the time of writing, eC does not yet support static access for members. Not being able to declare members statically has the unfortunate consequence that a data type used anywhere in the return types or parameters of functions, or the data declarations will not be allowed to be declared as static, even though is not required outside the *.ec* source file. Future improvements to the language and compiler should resolve this issue.

17 We haven't mentioned header files yet because eC mostly eliminate the need for them, although they are still available. The including process is still available for the few occasions in which it is useful to eC, and for interoperability with C libraries.

We already mentioned that following typical usage, and in line with the C++ defaults, structure members are public by default, whereas class members are automatically private. Just like global definitions, these can be changed either by group using the access control keyword followed by a colon (`public:` or `private:`) inside the class definition, or individually by preceding specific member definitions by the access keyword. Any member not part of an interface should be kept as private. An example follows:

```
class Class1
{
	// Method1 is private (default class access mode)
	void Method1()
	{
	}

	// data1 is private (default class access mode)
	int data1;

	// property1 is specifically marked as public
	public property int property1
	{
	   set { data1 = value; }
	   get { return data1; }
	}

public:
	// Method2 is public (class access is now public)
	void Method2()
	{
	}

	// property2 is public (class access is now public)
	property int property2
	{
	   set { data2 = value; }
	   get { return data2; }
	}

	// data2 is specifically marked as private
	private int data2;

private:

	// Method3 is private (class access private again)
	void Method3()
	{
	}
}
```

Typically, class data members are kept private whereas public properties are used to access them indirectly. Refraining from exposing data members publicly when building a shared library module, ensures binary compatibility with future versions of an interface, even if the data layout of a class change completely and if the user application derives classes from that modified class. That is made possible because eC amalgamates classes at program *initialization time* rather than at *compile time*.

Even though a member of a class (or structure) is private, it can still be accessed by other classes within the same module (we're referring here to target modules - dynamic libraries, the same will be true of *.ec* source file modules relating to static member definitions when they are made available). This is in great contrast with other object oriented languages such as C++. The only way to do so in C++ is by establishing complicated *friendship* relationships between multiple classes and possibly between classes and specific members (using the *friend* keyword), which in the light of Ecere philosophy adds unnecessary complexity. It otherwise alternatively results in writing superfluous methods to access said data members indirectly. In the context of a module often developed by a single developer, these practices are respectively counterproductive or have a negative impact on performance.

Therefore eC focuses on protecting private members strictly from being accessed outside of the module in which a class is defined, thus in effect encapsulating the entire implementation of an interface inside a module rather than single constructs such as a class. It is encouraged to group related classes together inside a single source file module (to benefit among other things from the static access control of global definitions, which will later be expanded to member definitions). This is again in contrast with the approach of having a source file for every class which is common in Java and C++ (where it most often also has a matching header file), resulting in a huge number of files difficult to manage.

Modules exporting public constructs whose interface makes use of other constructs in an other imported target module, *must* import that module *publicly* to ensure that the constructs required for the interface will also be available.

Other than controlling what is exported from a module, access control also affects the initializers of instantiations *within* that module. Private members can only be initialized by name (`identifier = [value]`), and are thus not part of the default unnamed initialization list. Thus the first default unnamed initializer in the instantiation of a class is reset to its first public member declared directly in that class, skipping any members declared in its base classes. The following example illustrates this:

```
class BaseClass
{
   int a;
public:
      int c, d;
}

// 1 is assigned to c, 2 is assigned to d
// a can still be initialized by name in same module
BaseClass baseObject { 1, 2, a = 5 };

class PublicDerivedClass : BaseClass
{
   int e;
public:
   int f, g;
}

// 3 is assigned to c, 4 is assigned to d
// a can still be initialized by name in same module
PublicDerivedClass pubDerivedObject { 3, 4, a = 5 };

class PrivateDerivedClass : private BaseClass
{
   int e;
public:
   int f, g;
}

// 3 is assigned to f, 4 is assigned to g
// a can still be initialized by name in same module
PrivateDerivedClass prvDerivedObject { 3, 4, a = 5 };
```

***Importing and working with multiple modules***

Things derive their being and nature
by mutual dependence and
are nothing in themselves.

*-- Siddha Nagarjuna*

So far we've written all our programs as a single *.ec* source file module. We've seen how to export an interface outside a shared library module using the public access mode, and we know that such shared library modules can be imported either publicly or privately. In this chapter we'll learn how to import modules (such as the Ecere runtime library) as well as how to build libraries.

However, first we will keep to a single target module but using multiple *.eC* source file module. As previously mentioned, any non static eC constructs (either private or public) will automatically be available across such modules upon importing that module, either directly or indirectly (e.g. by importing another module which in turn imports the needed module). The `import` keyword must be used followed by the name of the module between double quotes (typically stripped of the extension, although it could also contain the *.ec*). An example sharing a class (*TestClass*) and a function (*TestFunction*) across two modules follows. Other constructs such as variables, structures or enumerations would be available exactly the same way.

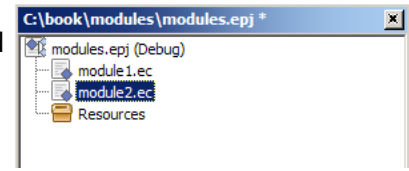
*module1.ec:*

```
import "module2"
TestClass object { };
class TestApp : Application
{
   void Main()
   {
      TestFunction();
   }
}
```

*module2.ec:*

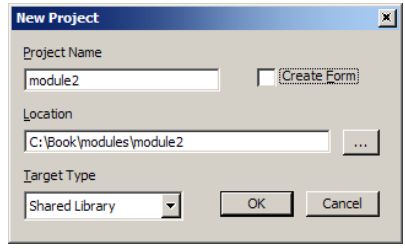
```
class TestClass
{
   TestClass()
   {
      printf("TestClass Constructed\n");
   }
};
void TestFunction()
{
   printf("TestFunction Called\n");
}
```

The import process of eC does not include the contents of the imported module inside the file which imports it, and thus is very different from the C `#include` preprocessor directive. Both modules must be added to the project, as displayed in this screen shot:




Only eC modules included in the project can be imported; the module name following the import keyword is being matched with each source file name (through *generated symbols* we will talk about on next page). Therefore regardless of the source directory hierarchy (e.g. a source module could be located in *src/common/supportModule.ec*), it would still be imported using only the file name, not the full path (*import "supportModule"*). With source file modules, the access control is not modifiable as all source modules are part of the same target module, and thus private constructs are available throughout. Thus naming conflicts might arise between them. They must be resolved by either renaming or, if possible, specifying one of them to be static to a single source module.

As you can see, importing a source module is quite trivial. We'll now learn how to implement the above using shared library modules. The shared library will consist of the *module2*, whereas the application will only consist of *module1*. First, let's create a new shared library project. We will call it *module2*.




We will add the source file *module2.ec*, which will be modified to export the declarations outside the shared library by making them public as such:

*module2.ec:*

```
public class TestClass
{
   TestClass() { printf("TestClass Constructed\n"); }
};

public void TestFunction()
{
   printf("TestFunction Called\n");
}
```

Upon a successful build, we will obtain a shared library in the target directory (*debug/* by default), either in the form of *module2.dll* on Windows, *libmodule2.so* on Linux or *libmodule2.dylib* on MacOS X. We will need to copy this module either to the project directory of our application (which will consist of only *module1.ec*) or to a location within our dynamic library path (*PATH* environment variable on Windows, *LD_LIBRARY_PATH* on Linux).

We can use the exact same code for *module1.ec*, and simply remove *module2.ec* from our previous project. However, if you chose to do so it is important to first ensure the *module2.sym* file is deleted from the intermediate object directory to avoid a conflict with our newly built *module2* shared library. It was previously generated when *module2.ec* was part of our main application project. When both the symbols generated from an eC module and a shared library of the same name exist, the eC module has priority. These files with the *.sym* extension are the results of a precompilation process across all source files of a target module and it is what enables the eC source file import mechanism.

You should now be able to build the main application project, which will make use of that second module we just built. If you previously built the project, it is recommended to use either the menu option Project / Clean or Project / Rebuild. At this moment, symbols are only regenerated when their associated source file is modified, and their potential dependencies are not automatically rebuilt. It is possible that other source files require to be rebuilt in order to match the new symbols of a modified source files. If in doubt, a project rebuild will rebuild all symbols and source files. Please also note that having a source file opened in the IDE which imports a dynamic library module will make use of it for enhanced code editing and designing support. It will be necessary to close all documents directly or indirectly importing the module should you want to modify or overwrite the dynamic module.

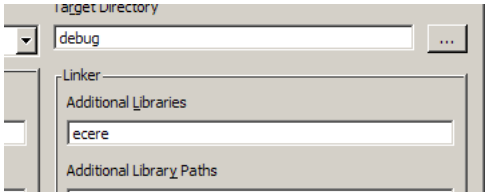
The modules imported this way are being loaded dynamically at program initialization. It therefore doesn't require any additional linking step and makes migrating portions of code from source modules into shared library modules very simple. There is no need for header files, and the programming interface is directly available within the runtime shared library. Only the exported constructs are available to the external user, while the rest remains invisible outside of the implementation. As we will learn in the *Advanced eC programming*, eC also supports loading these modules at runtime and using their functionality through *reflective programming* (which allows you among other things to instantiate a class or call a method on an instance by its name text string), thus it is very well suited for plug-in architectures.

When working with more than two levels of dynamic modules (e.g. an application importing a module which in turn imports another module), the import access mode of the deepest module affects whether it will automatically be visible in the higher level module or not. The default is *private* importing which prevents the higher level module from accessing the constructs of the deeper module without explicitly importing that module itself (it makes all imported public constructs private). Preceding import by the `public` keyword will ensure the dependent module is made available as well, which as we discussed in last chapter, is a requirement if the module interface makes use of any construct in that dependent module. We will illustrate this by using of the *Point* structure of the Ecere runtime library in a public interface:

```
// If public is omitted here we'll get an error:
// Public function making use of a private class
public import "ecere"

public void TestFunction(Point point)
{

}
```

An application requires the Component Object Framework which powers the eC languages if its project contains any source file with the *.ec* extension. The framework is packaged as a separate shared library (`ecereCOM.dll` or `libecereCOM.so`), and is also integrated inside the Ecere runtime library (*`ecere.dll`* or *`libecere.so`*) to prevent requiring two libraries. By default, a new project links with the Ecere runtime, but can be changed (in the project settings, see picture below) to link only with the COM framework if it does not make use of the runtime functionality. All other modules can be imported without actually being linked by the dynamic linker with the application making use of them.




It is also possible to make use of either the Ecere runtime library, the Ecere Component Object Framework, or any eC shared library module by linking to it *statically*, that is to include the library within the executable file rather than as an external shared library. However, because the framework is oriented towards shared libraries, a shared library version of of the module must first be built for the compiler to load module's available components.

This is usually not a problem, as most often both a static and dynamic version of the module is used for different configurations (e.g. debug, release). The *static library* selection in the *Target Type* option of the *New project* dialog (illustrated a few pages earlier) can be used to build a static library. The target type can also be used at any time in the *Project settings* dialog. After a static version of the module has been built, to link statically with a module, the import directive in the module making use of it must then be *followed* by the `static` keyword as such (which takes on a different meaning here than its access control usage):

```
import static "module2"
```

The name of the module must also be added to the *Additional Libraries* box seen in last screen shot, and its path must be added to the *Additional Library Paths* if it is not in a location specified in the relevant *Global Settings* dialog option.

In a similar manner, the `import` keyword can also be followed by `remote` which indicates classes specifically marked as remote declared in that module will be used remotely through the Ecere Distributed Component Object Model. A dedicated chapter in the section on network programming will cover this feature in details.

***Name Spaces***

The unnameable is the eternally real.
Naming is the origin
of all particular things.

*-- Tao Te Ching, verse 1*

As soon as we start working with multiple modules, building libraries or using a third party library, it is commonplace for naming conflicts to arise. The fact of names being used up is sometimes referred to as *name pollution*. The access control in eC allows to restrict the availability of identifiers to a single source file module or library module, and thus can prevent a great deal of name pollution. For this reason, constructs should always be kept as tightly knit as possible, with only the minimal required interface being exposed. However, name collisions can still occur between multiple libraries being developed by different development groups, or one might want to reuse particular meaningful identifiers in different contexts. Name spaces were designed to solve these issues.

The design of eC provides a way to avoid conflicts and regroup constructs pertaining to a certain type of functionality within name spaces. However, in order to preserve the syntax elegance as well as the development efficiency, eC does not require extra prefixes or special directives instructing which name spaces to check in order to use constructs defined within name spaces, unlike other programming languages. It features automatic name space resolution, effectively searching through all name spaces to resolve identifiers. The support of name spaces in eC is not yet completed, although what is currently implemented is fully functional. Among other things, a compiling option to disable that automatic name space resolution, as well as a warning system identifying ambiguous automatic resolution are planned.

By default, all constructs are declared within the *global name space*. It is possible to change that default by changing the *Default Name Space* in the project settings compiler options.


Constructs can be accessed from a specific name space by preceding them by its name space followed by the double colon (`::`), e.g. *`MyNameSpace::variable`*. A construct can also be declared to be in a specific name space the same way:

```
int MyNameSpace::variable;
```

Name spaces can contain other name spaces (sub name spaces). When specifying a name space at declaration, it is concatenated to the current name space.

It is possible to change the current name space by using the namespace keyword as such:

```
namespace MyNameSpace;
```

Anything following that directive will be declared within that name space until the namespace keyword is used again to change the current name space again. Note that using the namespace keyword does not concatenate the new name space to the current one, but always to the global name space (or the default name space specified in the compiler option). Thus a variable subsequently simply declared as:

```
int variable;
```

will be accessible as *MyNameSpace::variable*. Automatic name space resolution first attempts to resolve identifiers within the current name space.

Sub name spaces can also be specified at once, both for defining the current name space and for declaring constructs within it:

```
int MyNameSpace::SubNameSpace::variable1;

namespace MyNameSpace::SubNameSpace;

int SubSubNS::variable2;
```

Here the full identifier of variable2 will be:

```
MyNameSpace::SubNameSpace::SubSubNS::variable2;
```

Luckily automatic resolution allows to access it simply as *variable2*.

Name spaces apply equally to structures, classes, enumerations, functions and variables. For example a class could be defined in MyNameSpace as follows:

```
class MyNameSpace::MyClass { };
```

As was previously noted while we were discussing the access control topic, using name spaces in eC will result in mangling identifiers to take the name space into account. This results in an incompatibility with C, which doesn't support any kind of name spaces. Using the default declaration mode specifier overrides any current name space (thus placing a declaration in the global name space) and ensures C compatibility.

### *Units and conversion properties*

Can you coax your mind from its wanderings
and keep to the original oneness?

*-- Tao Te Ching, verse 10*

In the first section, we've covered all basic C data types to store integer and real numeric values. We've also mentioned the `typedef` keyword which allows to declare new type identifiers mapping to existing data types. eC further extends this concept by introducing the concept of *units*. A unit data type is any type which can be represented by a single numeric value, a fact which correlates them to the usage of `typedef` in C. Yet, just like structures and classes, methods and properties can be defined for them, thus allowing them to behave like completely new data types. A particularly useful feature is the notion of *conversion properties* establishing a relationship between any two data types.

A typical usage example of units is to implement measurement units such as meters and feet. Through conversion properties, the equivalence between a designated reference unit and other related units is automatically set up. Chain conversions by way of intermediate data types are also made possible. Nonetheless, conversion properties are not limited to unit data types, but also apply to more complex types such as structures and classes. Conversions between all these different types can be defined as well. We will also quickly revisit enumerations. Just like units, enumerations are internally represented by a single numeric value and both methods and properties can be defined for them.

Let's first look at the syntax for defining a new unit data type. The `class` keyword is used in the same way as for defining regular classes, along with a base class identifying a unit data type. The base class can either be an integer or a floating point type specifier, or another unit data type. As an example, we'll define a *`Distance`* class:[18]

```
class Distance : double { }
```

One advantage of such a unit definition over a `typedef` is that it can be imported across eC modules (just like regular classes), while a `typedef` must be placed within a header file to be included. Units cannot have data members, and thus the only initializer within their instantiation is the value to be assigned to a unit variable, for example:

```
Distance distance { 5 }; // distance is assigned 5
```

18 Some of the example classes we'll define in this chapter already exist in the Ecere runtime library, and thus would conflict if used while the "ecere" module is imported. The samples should also be linked with the *ecereCOM* library instead of *ecere*.

The previous definition is equivalent to the following:

```
Distance distance = Distance { 5 };
Distance distance = 5;
```

Note that although we declared *Distance* to use double precision floating point, we did not specify in which particular measurement unit we would store values it will hold. Ideally, the same unit will always be used to ensure all arithmetic operations make sense. Arithmetic can be performed just like on regular classes:

```
Distance a { 3 };
Distance b { 4 };
// sqrt is standard C math function for square root
Distance c = sqrt(a * a + b * b);   // c will be 5
```

Although eC does not yet support operator overloading, unit data types and automatic conversions can accommodate many use cases which would normally require operator overloading to implement in C++ for example. The simple example above could be implemented simply as a `typedef`, but adding methods and conversions would require making it a class in C++, and arithmetic operations would no longer be supported without overloading operators. In eC, unit data types always support arithmetic operations, and the only thing left to do is to establish the relationship between different units qualifying the same measurement.

Conversion properties, supported through all eC data types, allows an implicit conversion of one data type into another. The syntax for a conversion property is very similar to a regular property, except it does not have a name, only a data type. The specified type is the type for which we want to define a *to* and *from* conversion, through the `get` and `set` method, respectively. Only one of two related class needs to define the conversion for it to be available in both direction. Furthermore, anywhere a data type is expected, any other type from which exists a conversion to this expected type will be equally usable.

When multiple units of measurement are used, a *reference unit* must be defined. If a generic unit type such as *Distance* is defined, not invoking any particular measurement unit, the reference unit can be mapped to that generic unit type by defining a conversion property with neither a `set` nor a `get` method. In our example below we use meters as the reference distance unit. The empty conversion property to and from *Distance* defines *Meters* as directly mapping to the *Distance* unit. Consequently, all units deriving off *Distance* will be stored internally in this reference unit (*Meters*).

```
class Meters : Distance { property Distance { } }
```

Let's now define an imperial measurement unit, feet. We will need to define the conversion between feet and the reference unit (meters): a meter is 3.2808399 feet.

```
class Feet : Distance
{
   property Meters
   {
      set { return value * 3.2808399; }
      get { return this  / 3.2808399; }
   }
}
```

If we defined another *Distance* unit, such as *Centimeters*, we could operate on *Feet* and *Centimeters* even though we didn't define a direct conversion between them.

Note how the set method returns a value, rather than setting a data member (a unit class does not have any data member). This syntax might be improved in the future to additionally support the more intuitive syntax *this = value * 3.2808399*. The *value* identifier within the set method is the *Meters* unit to be converted *from*. The value returned from the set method is the value converted to *Feet*. In the get method, the this identifier represents the *Feet* value to be converted into *Meters*. This behavior of the properties set and get methods is shared among all kinds of data types stored as a single numeric value, which include the enumerations, units and the bit collection classes we will cover in the next chapter.

Now that we have defined two related distance measurement units, we can look at automatic conversions in action. A simple example is converting meters into feet:

```
Feet value = Meters { 5 };
printf("%f\n", value);
```

The output will be the expected 16.404200. Observe how using the instantiation syntax we qualify measurement values with a particular unit, such as *Meters* in this last example. These units are carried through arithmetic operations in much the same way they are in scientific calculations. When directly using numeric values, the expected data type dictates the unit of these non qualified values. For example, the following assignment is understood by the compiler to represent 5 meters plus 2 feet:

```
Feet value = Meters { 5 } + 2; // 2 means Feet { 2 }
```

However when no unit data type is expected and only one of the two addition operands has a specified type, the type of the other operand will be dictated by the operand specifying it:

```
double value = Feet { 5 } + 2;
```

This is equivalent to the following, making value equal to 7:

```
double value = Feet { 5 } + Feet { 2 }
```

As me mentioned before, internally feet will be stored as a `double` holding the measurement in the reference unit which we defined to be meters. However when assigning a *`Feet`* unit to a `double` or when using it where no data type is expected (such as in the last *`printf`* call), we will still get the numeric feet value. In order to obtain the internal value, a cast to the base unit type must be performed:

```
Feet value = Meters { 5 };
printf("%f\n", (Distance)value);
```

When using untyped destination results, it is possible for expressions to be ambiguous when mixing up units. Thus the compiler wouldn't know if the following `double` value should represent feet or meters:

```
double value = Feet { 10 } + Meters { 20 };
```

In this particular case the following warning will be issued:

```
operating on Feet and Meters with an untyped result, assuming Feet
```

Different classes of operators behave differently with units. We've pointed out that when using the + or – operator, operands without an associated unit type either take on the destination unit, if there is one, or otherwise the unit of the other operand. This would not make sense however for multiplication or division:

```
double value1 = Feet { 5 } / 2;
Meters value2 = Feet { 5 } * 2;
```

In both of these examples, the constant 2 does not have an implied unit, and simply divide or multiply the distance by 2.

Note that the initializer of a unit could be any expression resolving to the base data type of the unit (`double` in our examples). Thus the following syntax would be perfectly valid to specify inches as a fraction of a foot:

```
Meters value = Feet { 5 + 8 / 12.0 };
```

One advantage of defining classes implementing conversions inside shared library modules is that the compiler will be able to perform any constant conversion at compile time. The compiler currently cannot perform compile time conversions for those defined in a non previously compiled module, as it will require a code interpreter.

We've seen conversion properties in action with units. Let's use them with other eC object oriented constructs, starting with structures. We'll use polar and Cartesian coordinates as an example.

```
#include <math.h>[19]
struct CartesianPoint { Distance x, y; };
struct PolarPoint
{
   double angle;        // angle in radians
   Distance distance;
   property CartesianPoint
   {
      set
      {
         angle = atan2[20](value.y, value.x);
         distance = sqrt(
            value.x * value.x + value.y * value.y);
      }
      get[21]
      {
         value.x = cos(angle) * distance;
         value.y = sin(angle) * distance;
      }
   }
};
```

With these structures and conversions in place, a *PolarPoint* can be provided in lieu of a *CartesianPoint*, and vice versa. The following example demonstrates:

```
void HandleCPoint(CartesianPoint point) { }
void HandlePPoint(PolarPoint point) { }
void Test()
{
   HandleCPoint({ 3, 4 });
   HandleCPoint(PolarPoint { 3.14 * 53.1 / 180, 5 });
   HandlePPoint(CartesianPoint { 3, 4 });
}
```

19 We exceptionally need to include <math.h> here because by default cos & sin functions expect an *Angle* unit type defined in the ecere module.

20 atan2 is a standard C function returning the arc tangent from the two sides at right angle of a triangle

21 Note how *value* is set to return the *CartesianPoint*, as structure conversion properties require. A more intuitive return { x, y }; syntax might be implemented in the future.

Defining conversion properties for regular classes implies dynamic memory management, and is thus not used frequently. In fact there is currently limitations to using conversion properties along with regular classes, only the `get` method is usable. In practice, this means that at the moment one can only *return* a regular class, either from another regular class or from another construct such as a structure. It also prevents two way conversions from functioning with the current state of things. These limitations are mainly due to the fact that conversion properties are not as useful with regular classes as they are with unit data types for example, thus efforts have not yet been focused on supporting them fully. The following example illustrates what is already supported:

```
class ClassOne
{
   public int a, b;
}

class ClassTwo
{
   public int a, b;

   property ClassOne
   {
      // Our sample conversion between ClassOne & ClassTwo
      // consists in merely matching members a and b
      // This new instance here will need to be freed
      get { return ClassOne { a, b }; }
      // A current compiler limitation prevents this set
      // method from being used properly
      set { a = value.a; b = value.b; }
   }
}

void DealWithOne(ClassOne object) { delete object; }

class TestApp : Application
{
   void Main()
   {
      ClassOne a { };
      ClassTwo b { };

      DealWithOne(a); // This will delete a
      DealWithOne(b); // This will delete a new ClassOne

      delete b; // b still needs to be deleted
   }
}
```

Let's now revisit enumerations, and see how we can add methods and properties to them as well. These can be placed after the enumeration values, terminated by a semicolon. The following example converting between classical Chinese elements and their associated planets demonstrates how it's done:

```
enum Element : byte
{
   wood, fire, earth, metal, water;
   property Planet
   {
      get
      {
         switch(this)
         {
            case wood:  return jupiter;
            case fire:  return mars;
            case earth: return saturn;
            case metal: return venus;
            case water: return mercury;
         }
      }
      set
      {
         switch(value)
         {
            case jupiter: return wood;
            case mars:    return fire;
            case saturn:  return earth;
            case venus:   return metal;
            case mercury: return water;
         }
      }
   }
   property Planet planet
   {
      get { return (Planet)this; }
   }
   void PrintValue() { printf("%d\n", this); }
};

void Test()
{
   Element element = mercury;
   Planet planet = fire;
   // A regular property get can be used with enums:
   planet = element.planet;
   element.PrintValue(); // Methods can be used too
}
```

As previously mentioned, conversions between different kinds of data types are also possible. Just as an example, we'll implement a conversion to and from our *Distance* unit in our *PolarPoint* structure from earlier:

```
#include <math.h>

struct CartesianPoint { Distance x, y; };
struct PolarPoint
{
   double angle;        // angle in radians
   Distance distance;
   property CartesianPoint
   {
      set
      {
         angle = atan2(value.y, value.x);
         distance = sqrt(
            value.x * value.x + value.y * value.y);
      }
      get
      {
         value.x = cos(angle) * distance;
         value.y = sin(angle) * distance;
      }
   }

   property Distance
   {
      get { return distance; }
      set { distance = value; }
   }
};
```

Now let's say we'd like to compute the distance in feet between the origin and a point at 300 meters along the x-axis and 400 meters along the y-axis, we can simply write the following line of code, making use of multiple conversions:

```
Feet d = (PolarPoint)CartesianPoint { 300, 400 };
```

The result will be ~1640.42 feet (500 meters).

***Bit collection classes***

The Tao can't be perceived.
Smaller than an electron,
it contains uncountable galaxies.
*-- Tao Te Ching, verse 32*

Bit collection classes are the last kind of data type available in eC which we have not yet covered. Like structures, they can hold multiple data members, yet the entire class can be used as an integer value. They provide a very elegant manner to manage flag collections stored in integer types for example, which typically require numerous cumbersome macros in C (using `#define` preprocessor directives). With these, they further remove the need for binary logic operators, and instead provide a much more intuitive way of modifying or checking the values of specific bits. The syntax for defining bit collection classes is inspired from the declaration syntax of C structures bit fields specifiers. Bit fields specify exactly how many bits one particular data member should take. For example the following will reserve only a single bit for each boolean data member:

```
struct CarOptions
{
   bool electricLocks:1, airConditioned:1;
   bool automaticTransmission:1;
} options = { true, true, false };
```

C bit fields however do not allow using the defined structured as an integral value, which makes it incompatible with the unsigned integer type used for flag collections. The following code is not valid:

```
uint a = options; // incompatible expression: options
```

The bit collection classes of eC solve this problem by remaining integer data types, yet letting both the size and position (in number of bits) of every member be specified. Specifying the position allows the bit order to be different from the declaration order, and thus a specific memory layout can be achieved while keeping the desired initialization order. The total number of bits for all the data members can not exceed the number of bits of the integer size the bit collection class is derived from (which can be any integral type, usually either `uint` or `uint64` for more storage). However it is possible for multiple data members to overlap, as union data members would overlap. A bit collection class definition equivalent to the above structure would be:

```
class CarOptions : uint
{
   public bool electricLocks:1, airConditioned:1
   public bool automaticTransmission:1;
}
```

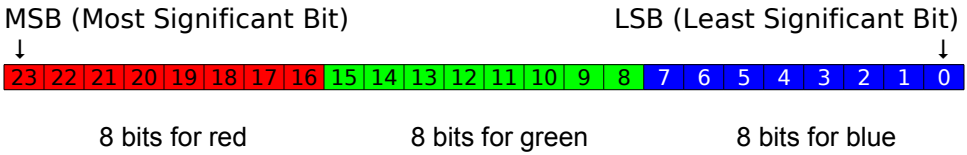
Note the similarity between the bit collection class definition and a unit definition, the only difference being that bit class definitions contain data members (with bit field specifiers). In this first example we only specified the size to be taken by each data member (1 bit). Like other constructs, bit classes use the eC instantiation syntax. Thus the following are all valid uses of our *CarOptions* bit class:

```
CarOptions options { true, true, false };

CarOptions options = { true, true, false };

CarOptions options;
// options is undefined until it is initialized:
options = { true, true, false };

CarOptions options { }; // All members are zeroed

// Non initialized instantiation members will 0:
CarOptions options { electricLocks = true, true };
```

Each flag can then be individually accessed like a regular data member, while *options* can still be used with integral types:

```
bool electricLocks = options.electricLocks;
uint a = options;
options.automaticTransmission = true;
options.electricLocks ^= true; // Toggle with XOR
```

We'll apply bit collection classes to color representation for our next examples, where we will specify the bit position as well. Our objective will be to define a *Color* bit collection class[22] which has the following memory layout:




Since 8 bits are used to represent each red, green and blue component, the intensity value will range from 0 to 255. The default memory layout in bit classes, following the declaration order, is from LSB to MSB. Therefore in our previous example, *electricLocks* would be stored in LSB (bit 0), *airConditioned* would be stored in bit 1, and so on. We'd like to keep the familiar RGB initialization order, which does not match the memory layout going from blue to red, so we will need to explicitly specify bit positions.

22 Similar Color classes are defined in the Ecere runtime library, therefore the "ecere" module should not be imported or linked with to try out the examples in this chapter.

The number after the second colon is the starting bit position for each data member:

```
class Color : uint
{
public:
   byte r:8:16;   // 8 bits, starting at bit 16
   byte g:8:8;    // 8 bits, starting at bit 8
   byte b:8:0;    // 8 bits, starting at bit 0
}
```

We can then use our *Color* class to define the background color of this book's cover as such:

```
Color color { 53, 98, 133 };
```

And each member can be accessed individually, just like a regular data member:

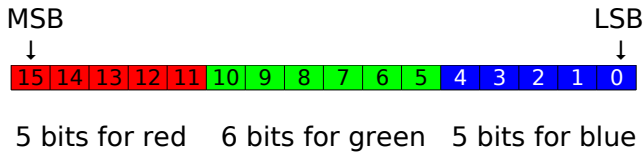
```
int r = color.r;
color.r = 100;
```

It is also possible to use it as an integer. For example the following call prints it in the hexadecimal syntax typical of HTML:

```
printf("#%6X\n", color);
```

Note that if neither size nor position is specified, the size will be that of the data type qualifying the member (e.g., 8 bit for bytes), and the bit position always increases by the number of bits taken by each member.

Just like other eC object oriented constructs, bit collection classes also support conversion properties. We will demonstrate their usage to perform conversions between our *Color* class and a common 16 bit color format. Here is the layout for the "565" RGB format we will convert with:




With this format the intensity for green has one more bit, thus supporting a range of 0 to 127, while red and blue intensities go from 0 to 63. We perform the conversions by shifting the bits of each color component so as to always keep the highest bits, by 3 bits (8-5) for red and blue components, and by 2 bits (8-6) for green.

Here is our *Color565* class implementation:

```
class Color565 : uint16
{
public:
   byte r:5:11;   // 5 bits, starting at bit 11
   byte g:6:5;    // 6 bits, starting at bit 5
   byte b:5:0;    // 5 bits, starting at bit 0
   property Color
   {
      set
      {
         return Color565
         {
            value.r>>3, value.g>>2, value.b>>3
         };
      }
      get { return Color { r<<3, g<<2, b<<3 }; }
   }
}
```

Together our *Color* and *Color565* classes can be used to perform conversions very easily:

```
// Starting colors
Color color { 53, 98, 133 };        // 0x356285
Color565 color565 = { 6, 24, 16 };  // 0x3310

// Automatic conversions
Color    color24 = color565;
Color565 color16 = color;
```

Just to illustrate how bit collection classes can simplify code, the conversions would look like this without them:

```
uint color24 =
      ((((color565) & 0xF800) << 8) |
       (((color565) &  0x7E0) << 5) |
       (((color565) &   0x1F) << 3));
uint color16 =
      ((((color) & 0xF80000) >> 8) |
       (((color) & 0x00FC00) >> 5) |
       (((color) & 0x0000F8) >> 3));
```

These numbers had to be computed from the binary layout of both classes. Conversions through bit collection classes greatly reduces the risk for errors by automatically performing the work, based on the layout specified in the class definitions and the simple conversion property code. Their usage is also a lot more elegant.

# Section 3

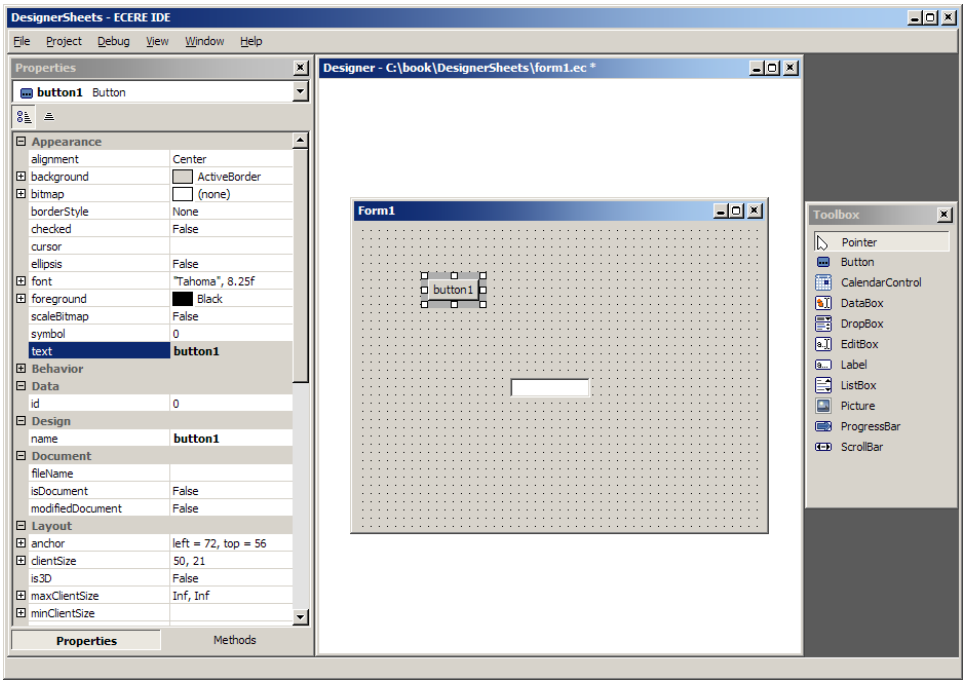
## Building Graphical User Interfaces

The form designer and property sheet
Static controls: labels and pictures
Events, buttons and option boxes
Inputting data through edit boxes
Displaying message boxes

### *The form designer and property sheet*

The Ecere development environment provides a powerful, extensible visual editor system for designing objects of particular kinds. Bundled within the Ecere user interface system, a *form designer* allows visual edition of *Window* classes and objects. The *Window* class is the foundation of all Ecere graphical user interface elements: the main application window, all dialogs and all controls are all derived from a *Window* class.

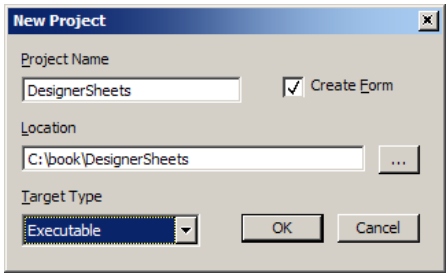
The object designer works hand in hand with the *Property/Methods Sheet*, which lists all available properties for the class or object being edited. Together with the object designer, they form the basis for the Ecere Rapid Application Development (RAD) visual programming system. Another element of the RAD system is the toolbox which can be found on the right-hand side of the IDE. The toolbox shows available classes of instances which can be added to the class being edited by simply dragging them with the mouse onto the designer's visual representation.



Furthermore, all these are integrated and automatically synchronized with the regular source code text editor. Whenever a modification is made in the source code, it will be reflected in the object designer as soon as it gets focus. The opposite is also true: modifications made from the object designer or property sheet will be visible in the code editor when it gets the focus back. In contrast with other RAD systems, all the code generated by the object designer and property sheet system *is* meant to be user-editable, and easily so. It does not generate any caution comment or reserved code areas, but rather strictly compact code directly associated with the visual modifications.

Let's create our first application with a graphical user interface. We'll use the New Project dialog again, but this time we'll leave the *Create Form* check box checked:




Right after pressing the OK button, we're taken to the form designer which contains an empty form. In the screen shot on the previous page, we simply dragged an edit box and a button from the toolbox onto the form. The F8 accelerator key can be used to toggle between the form designer and the code editor. The code for an empty form project will look like this:

```
import "ecere"

class Form1 : Window
{
   text = "Form1";
   background = activeBorder;
   borderStyle = sizable;
   hasMaximize = true;
   hasMinimize = true;
   hasClose = true;
   clientSize = { 400, 300 };
}

Form1 form1 {};
```

You can see that the Ecere runtime library is automatically imported when creating a new form project; it is required since that is where the user interface functionality is located. Then a *Form1* class is defined, inheriting from the *Window* base class. The properties set by default will assign some size to it and will equip it with a title bar as well as the typical maximize, minimize and close buttons. Then we have a global instantiation of the *Form1* class.

This form application, as it is generated by the New Project dialog, is ready to be ran. It will create that form previewed in the form designer, and will execute until the form is closed. Recalling the first chapter of this book and its *Hello, World* example, one might wonder exactly where the entry point for this application and its execution code are. The answer is that the Ecere runtime library contains a default *Application* class which will be used if none is defined in the application. The *GuiApplication* class (derived from *Application*) provides the basis for building GUI based applications. Applications making use of the user interface constructs of the runtime library should either derive their application class from it, or simply not define any *Application* class, in which case *GuiApplication* will be used.

*GuiApplication* takes care of overriding the familiar *Main* method, making the application run as long as a window is still up on the screen. Note that if we comment the instantiation of the *Form* class, the application will thus start and exit right away. *GuiApplication* also provides three new virtual methods which can be overridden to perform operations at specific times:

```
virtual bool Init(void);
```

Executes when the application initializes.

```
virtual bool Cycle(bool idle);
```

Executes at every iteration of the main user interface loop, *idle* being true if no input has been processed within this cycle.

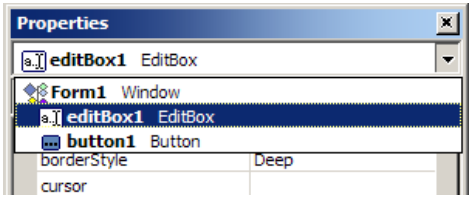
```
virtual void Terminate(void);
```

Executes before the application terminates.

*GuiApplication* also encapsulates various and functionalities pertaining to the application as a whole, some of which we will cover in the rest of the book.

Let's go back to the form designer. As it first appears when creating a new form or project, the form itself is selected (as visually indicated by the eight white resizing boxes and gray moving border surrounding it). The property and method sheets will always reflect the current selection, listing the properties or methods for that particular class or object. The form itself is the class being edited: it matches an actual class definition (derived from Window) in the source file opened in the code editor. When dragging controls from the toolbox onto the form, instances of these control classes are added to the class. These too can be selected, and will show up in the property sheet as well. In the drop box at the top, you fill find the form class first and the instances indented right under it:




Whereas modifying the properties for the class affect the default properties value for the class, modifying the properties for the instances will affect the instantiations' member properties initializers. When moving the cursor through the code editor, the instantiation or class inside which the cursor is located will automatically get selected. Similarly, selecting a class or instance from the property sheet or designer will move the cursor in the code editor to the associated code. Only classes with an associated class designer (such as the Window classes) will be found in the property sheet. When more than one class exists within a single source file, a different class can be selected through the drop box selector. Note that it is possible to drag instances from the toolbox directly inside an appropriate class within the code editor.

When dragging a control inside a form, the control is automatically associated to the form through both the *parent* and by extension the *master* properties. The *parent* window of a control is the window within which its drawing is contained. The control is thus said to be a *child* of that window. The *master* window is the window to which it belongs and which will receive all notifications (e.g. a button click notification) from the control. The control is thus said to be a *slave* of that window. When not explicitly set, the master window is set to the parent window. Both the *parent/child* and *master/slave* relationships are very important concepts in the Ecere GUI system and have many other implications, some of which will be covered in the upcoming chapters. Using the form designer, it is possible to reassign a control to another parent, for example another control within the form. You can do so by dragging it onto another control within the form. You will find its drawing area to be restricted to that new parent control.

The same thing could be done by entering the *name* of that other the control in the parent property. Every class and instantiation in the designer will have that special *name* property, which is strictly a *design mode* property and can be found within the *Design* collapsable property group. It uniquely identifies the class or instantiation, and will also be reflected in the identifier used within the code.

The property sheet, designer and code editor together offer many alternative ways to design and edit a class and its components. A selection in the designer can be moved by simply dragging around the object or the gray border surrounding it. It can be resized using the eight white boxes placed on the border. The property sheet lists properties such as *size* and *position* (within the *Layout* group), where the coordinates can be edited either together or individually by expanding the property (of a structure data type). When a property is selected in the property sheet, but the designer is the active window, starting to type will automatically activate the property sheet and typed text will go in the currently selected property. For example, by default the *text* property is selected and thus typing text from the form designer will modify the text displayed on the control.

Many properties are often connected together in a complex manner, and thus the order in which they are initialized might matter. For example, changing its text will automatically affect the size of a button. Similarly, selecting a different font will also update the button to be properly resized. All modifications, whether from the property sheet, designer or code editor, all have immediate synchronized visual feedback in the designer.

Feel free to explore the various controls and their properties available to get an overview of the Ecere GUI system and the form designer. Starting from the next chapter, we will start looking at some of the available controls.

# The Ecere Tao of Programmming

In Chinese philosophy, the *Tao* (道, *pinyin: Dào*) refers to "The Way of Nature". In his *Tao Te Ching*, *Lao Zi* illustrates the importance of harmony with the natural order of the Universe.

The Ecere Software Development Kit aims to provide a more natural approach to programming computers.

Ecere introduces eC, a language bringing together the efficiency of native C programming along with the simplicity, expandability and maintainability of modern object oriented paradigms. Most importantly, eC features a very elegant syntax, yet maintains a maximum compatibility with C.

The SDK features a powerful cross-platform Graphical User Interface toolkit, a graphics and image manipulation system, a networking library, and a 3D graphics engine running on top of both OpenGL and Direct3D.

This book teaches programming using these powerful tools from the ground up. Learn the concepts and syntax which eC shares with C and other derived languages (C++, Java, C#, ...). Learn the concepts and applications of object oriented software development. Learn how to build graphical user interfaces, manipulate and display graphics, develop networking applications. Learn 3D graphics programming.

Study in-depth many programming samples, including a full-featured 3D chess game with artificial intelligence and networking capabilities.

Learn a whole new programming philosophy valuing simplicity, elegance, and performance. Learn how to program the Ecere Way.

The Ecere Software Development Kit is included with this book, and contains all you require to start developing your own royalty-free applications (commercial or not) for Windows, Linux and Mac OS X.

## About the author

**Jérôme Jacovella-St-Louis** has been developing the Ecere SDK since 1996. He designed the eC language and is currently responsible for most developments in the SDK. Jérôme previously worked on InterMAPhics geotracking products for Gallium Software. He resides in Gatineau, Québec, Canada where he works for Ecere Corporation as Chief Technology Officer.