**From source to executable**

A C/C++ program is just a simple text file, and in principle you don't need anything complicated to write this. Even something very basic like notepad (on Windows) or nano (on Linux) could be used, but usually something more advanced (with syntax highlighting for example) is used.

The *compiler* is what transforms such so-called source files into a binary file which can be executed on your computer. This is different in languages such as Matlab or Python, which read the source file and execute what they read. The steps to compile a program are usually subdivided in two main classes:

1. compilation
2. linking

Usually the compiler can do both, using another program (you guessed it, the *linker*) to accomplish this second step.

The first phase, compilation, analyses a source file and produces machine code for this source file. The result is usually called an *object* file. Such an object file can not be executed yet, for this some other startup code and possibly other object files have to be added to it. And that's what the linker does: it takes a number of these object files as input, and creates an executable of them, possibly linking it to other libraries (existing collections of such object files) like the math library for example.

**Programming environments**

In a Linux environment (e.g. on the VSC cluster), you could transform a few simple source files into an executable with these commands (gcc is the GNU C Compiler):

gcc -c -o file1.o file1.c # -c specifies 'compile only', so don't link

gcc -c -o file2.o file2.c # -o specifies the output file name

gcc -o myexecutable file1.o file2.o # The .o files are object files, this actually links everything

You could also do the compiling and linking in one line:

gcc -o myexecutable file1.c file2.c # Using the .c files here

While the one-line approach may seem simpler, this causes everything to be compiled every time you change something, which may test your patience for larger programs. This is why usually object files are created, so that files are only recompiled when necessary. If not, an existing object file is used.

To automate the process of creating an executable based on some source files, several things have been created. One of the more basic things in the Linux/Unix world is the use of *makefiles*. Basically, such a makefile specifies which files are needed to create a specific output file, and the make command then uses this information to generate all files needed, ending with the creation of your executable.

Even creating such a makefile and figuring out which files depend on which can easily get very complex. For this reason other tools exist which start from another kind of configuration file, analyze this together with your source code, and finally produce a makefile to automate the building of the executable. Personally I like to use the CMake system for this, because it not only supports the generation of makefiles, but also other types of build configurations.

Apart from using all these kinds of text files for configuring your build, there are also more elaborate programming environments which provide an advanced editor to work on your source code and provide a nice graphical user interface to control the compilation and linking steps. These so called Integrated Development Environments (IDE) also automatically figure out what needs to be recompiled if a source file changes. On MS-Windows systems, the best known IDE is Microsoft's own Visual Studio, which actually supports other languages than C/C++ as well.

So while the basic idea is always the same -- transforming a text file with source code into an executable using compilation and linking steps -- things rapidly become different depending on the platform you want to use, what editor you'd like to use, what build system, etc. And of course, there's no single best solution for all this, just many people preferring just as many things.

**The C language**

The roots of the C++ language lie in the C language, and knowing some basic things is definitely useful. In C, you use functions (either from another library or your own) to accomplish tasks. For example:

int myFunction(int param1, double param2)

{

int returnValue;

/\* Some operations to calculate returnValue \*/

return returnValue;

}

Functions have a return type (int in this case, an integer value), a name, and parameters (an integer and a double precision floating point value). Here, returnValue will only exist while the function is being executed. A new call to the same function cannot assume that the previously calculated value is still known. The return statement ends the function execution and specifies what value the function returns. The *function body*, the actual code of the function is found between the curly braces { and }. You'll find these braces a lot in C/C++ code, they define a block of code that belongs together. They also define a *scope*: variables defined in one block, are no longer known when leaving the block.

The returnValue variable is stored in memory local to the function (sometimes referred to as 'on the stack'). In C, all such local variables have to be declared at the top of the block, before any real code. This is no longer necessary in C++, in fact to help avoid bugs it's good practice to only declare the variable name when it's going to be needed. Variable names must be known before using them.

Function names also need to be known when calling them. For example, the following code won't work:

int myFunction1(int param1)

{

return myFunction2(param1);

}

int myFunction2(int param2)

{

/\* some code here \*/

}

The compiler goes through the code from top to bottom, and when the function call myFunction2(param1) is executed, the name myFunction2 is not yet known. This will result in a compilation error. This can be mitigated by telling the compiler that the name myFunction2 exists and is some kind of function, before defining myFunction1:

int myFunction2(int param2);

int myFunction1(int param1)

{

return myFunction2(param1);

}

int myFunction2(int param2)

{

/\* some code here \*/

}

In fact, we don't even need to have the actual code for myFunction2 in the same source file: the actual implementation of that function is only needed at linking time, when an actual self contained executable needs to be created. The compiler itself will be satisfied knowing that such a function exists somewhere and knowing how it needs to be called.

This is why in most C/C++ files you'll see lines starting with #include at the top of the file. They just contain the names and parameters/return type (called the *signature*) of a whole lot of functions. The include statement just inserts the contents of one file in the other, so we could have created a file myheader.h with the following line:

int myFunction2(int param2);

And then in our original source file we could have written

#include "myheader.h"

int myFunction1(int param1)

{

return myFunction2(param1);

}

You'll also see < and > being used in such include lines instead of ". This has to do with the way the compiler looks for include files.

Apart from using simple types like int and double for example, you can also create new types by grouping them in something called a struct. First you have to define what the struct looks like,

struct myVector

{

double x, y, z;

};

and then you can use this in various ways, for example as a parameter of a function:

double getLengthSquared(struct myVector v)

{

return v.x\*v.x + v.y\*v.y + v.z\*v.z;

}

Parameters are passed by creating a copy, so if a function like this would be created:

/\* 'void' means no return type or no arguments (in C) \*/

void clearVectorBAD(struct myVector v)

{

v.x = 0;

v.y = 0;

v.z = 0;

}

and called like this:

void testFunction(void)

{

struct myVector v2;

clearVectorBAD(v2);

}

it wouldn't actually do anything, v2 would still be initialized to random values. You can overcome this by obtaining the memory address of v2 using the & operator (the *reference* operator). In the clearVector function we have to specify that we're actually getting a *pointer* to some memory, and we have to *dereference* it using the \* operator:

void clearVector(struct myVector \*v)

{

(\*v).x = 0;

(\*v).y = 0;

(\*v).z = 0;

}

void testFunction(void)

{

struct myVector v2;

clearVector(&v2);

}

The (\*v).x syntax is rarely used, usually an -> arrow is used to point to the actual memory location:

void clearVector(struct myVector \*v)

{

v->x = 0;

v->y = 0;

v->z = 0;

}

Because this time the address of v2 is copied and its contents are modified, the function will do what we want it to do.

I've already mentioned that variables created 'on the stack' are destroyed when exiting the block containing them. If we want to use more persistent memory (or if we need a large amount of memory), we have to allocate it dynamically. When we no longer need the memory, we need to release it to the system again, otherwise we're creating what is called a memory leak. If such a leak occurs in a 'for' loop for example, this could make your program terminate too soon because it runs out of memory.

When allocating such memory (often referred to as 'on the heap'), we get an address to a memory block, similar to an address obtained by the & operator. To allocate memory, the function malloc needs to be used which needs an argument specifying how many bytes should be allocated:

struct myVector \*v;

v = (struct myVector \*)malloc(sizeof(struct myVector));

clearVector(v);

free(v);

The sizeof(struct myVector) specifies how much memory we need for the struct we defined ourselves. The thing in front of malloc between parentheses, is called a type cast. By default, the malloc function has void \* as return type, indicating a memory address, a pointer, of no particular type. Since v is of type struct myVector \*, the compiler won't like this mismatch and will complain. By adding the cast, we're telling the compiler to interpret the void \* as a struct myVector \*, and it won't complain anymore.

As said before, when we don't need the allocated memory anymore, the memory must be returned to the system. This is done using the free function. The free function takes one argument of type void \*. This also doesn't match the type of v which is struct myVector \*, but since it's *less* specific, the compiler won't complain.

**The C++ language**

**Member functions**

In C, data is usually organized in 'structs' and calculations based on the data, or manipulation of the data, is done using functions. So in the example above, there's a function getLengthSquared to calculate some property for data in a struct myVector object.

Instead of defining a function which works with the data of some structure, in C++ we can define a function *inside* this struct:

struct myVector

{

double getLengthSquared()

{

return x\*x + y\*y + z\*z;

}

double x, y, z;

};

Now we've defined a function *inside* the structure, often called a *member function*, which operates on the data of a specific vector. A short example in which values are initialized and the square of the length is calculated would look like this:

struct myVector v;

v.x = 1.0;

v.y = 2.0;

v.z = 3.0;

double l = v.getLengthSquared();

So instead of using a function to use data from v, we're now asking v to tell us what it's own length squared is. This is an important step towards object-oriented programming: instead of just grouping data, a structure can now know how to operate on its own data.

As an aside, notice that the 'double l' is only declared at the time it is needed. Placing something like 'double l' after an assignment ('v.z = 3.0' for example) is illegal in C, where all the type declarations have to be at the top of a curly brace block. Also, in the line that says 'struct myVector v', typing the 'struct' keyword is no longer really necessary. After defining the layout of the structure, the C++ compiler now knows to recognize myVector as if it were a new type on its own, like an int or a double.

In the definition of the structure, we've placed the code of getLengthSquared inside the struct. While this is allowed, this is usually only done if the code is short. The compiler also doesn't need the code at that point, the actual implementation is only needed when linking the program. The structure definition then could look like this:

struct myVector

{

double getLengthSquared();

double x, y, z;

};

To get a working program, it necessary of course to have the implementation somewhere. This is then specified in the following way:

double myVector::getLengthSquared()

{

return x\*x + y\*y + z\*z;

}

Using the myVector:: prefix, the compiler knows that it's the myVector member function that we're defining. The compiler also needs to know that myVector exists of course, so if the struct is defined in a header file called myVector.h, the file containing the implementation of getLengthSquared will have a line

#include "myVector.h"

at the top.

Although we've added our first member function to the structure, the previous getLengthSquared(struct myVector \*v) function is still perfectly usable. It doesn't even need to have a different name, the compiler can figure our to which function we're referring from the context in which it is used. But often we don't want other functions to operate on (or even change) the data in our structure directly. For this reason, it's possible to tell the compiler that certain things in the structure are private, only accessible to object which are of the myVector type:

struct myVector

{

double getLengthSquared();

private:

double x, y, z;

};

Everything specified below the private line, is only accessible to member functions of myVector. As you might have guessed, the opposite of the 'private' keyword, is 'public'. In a struct, by default the data and member functions are public.

**From structs to classes**

In C++, instead of using structs usually classes are used. A class is just a struct in which everything is private by default. So the following would be completely equivalent to the struct definition above:

class myVector

{

public:

double getLengthSquared();

private:

double x, y, z;

};

Apart from the fact that a class is a concept used in many object-oriented languages, I don't think there's a specific reason that usually in C++ classes are used instead of structs.

Special kinds of member functions are *constructors*, which are called when an object of the type is created. In the example above, it could be helpful it the vector components would be initialized to zero when an object is created. This could be done by changing the class definition to

class myVector

{

public:

myVector();

double getLengthSquared();

private:

double x, y, z;

};

and adding a member function implementation like this:

myVector::myVector()

{

x = 0;

y = 0;

z = 0;

}

In general, a constructor can not have a return type and must have the same name as the class itself. When such a myVector object is defined in your code somewhere, the constructor is automatically used:

myVector v;

will create a zero-vector. There also exists a special function which is called when the object is being cleaned up, called a *destructor*. In this example the destructor would have the special name '~myVector()'.

It would also come in handy if we could just initialize all of the components of a vector when declaring the variable. For this, we could use a constuctor myVector(double a, double b, double c) with implementation

myVector::myVector(double a, double b, double c)

{

x = a;

y = b;

z = c;

}

And fortunately, having two constructors is no problem, the C++ compiler will figure out which one to use from the context. So in the code

myVector v1;

myVector v2(1.0,2.0,3.0);

the first line the constructor without arguments is used, and in the second line the constructor with three arguments is used. This is a nice example of the C++ support for *function overloading*. Functions with the same name can exist, as long as they accept different (different in number or different in type) arguments. Based on the number and type of the arguments used, the compiler will choose one specific version.

Arguments to functions are still passed by creating a copy of the variable, so just like in C, if the contents of the variable need to be changed in some function, a pointer to the object needs to be passed. For example, suppose we add a set function to myVector,

class myVector

{

public:

myVector();

myVector(double a, double b, double c)

double getLengthSquared();

void set(double a, double b, double c)

{

x = a;

y = b;

z = c;

}

private:

double x, y, z;

};

where we've just added the implementation to the class for brevity. The following function could then be used to set the contents of a vector that was passed using the & operator:

void someFunction(myVector \*v)

{

v->set(9.0, 8.0, 7.0);

}

So this could be used in the following way:

myVector v;

someFunction(&v);

In C++, one could also define a function as follows

void someFunction2(myVector &v)

{

v.set(9.0, 8.0, 7.0);

}

and do

myVector v;

someFunction2(v);

The function can be used in the same way as if a copy were going to be made, but because the & is present in the function definition, the compiler knows that the address of the variable should be copied and not the entire variable itself.

In C, the functions malloc and free have to be used to allocate memory in a dynamic way. While these can still be used in C++, they do not cause a constructor or a destructor to be called. For this reason, in C++ it's common to use the keywords new and delete instead, e.g.

myVector \*v = new myVector(5.0, 6.0, 7.0);

// do something with v

delete v;

**Inheritance**

Suppose we've got a class for a two-dimensional vector.

class myVector2D

{

public:

double getLengthSquared() { return x\*x + y\*y; }

double x, y;

};

For this simple example, everything is just public and the code of the function is just placed inside the class definition. If at one point a three dimensional vector would be needed as well, which would also need x and y data fields, we can use *inheritance* to define this new class:

class myVector3D : public myVector2D

{

public:

double getLengthSquared() { return x\*x + y\*y + z\*z; }

double z;

};

The first line says that the myVector3D class has myVector2D as a *base class* and adds a data entry z. Both classes can have the getLengthSquared member function, that's no problem. If there's some ambiguity as to which precise function should be used, a prefix like 'myVector2D::' or 'myVector3D::' can be used to specify which one. For example, in the 3D version, the function could also have the following implementation:

double getLengthSquared() { return myVector2D::getLengthSquared() + z\*z; }

**Virtual functions**

If we create an object of type myVector3D, it actually has two getLengthSquared functions. One just uses the x and y components, and the other uses the z component as well. While this can be useful in some cases, it's also possible that we'd prefer that only one single implementation is present, depending on the exact type of the object. To do this, the function can be declared as *virtual*. In principle it has to be virtual in both classes, but if it was specified already in the base class, the compiler automatically re-uses this in the other class.

These virtual functions are a very powerful thing. To illustrate their use, let's add another base class, to describe an arbitrary vector:

class myVector

{

public:

virtual double getLengthSquared() { return 0; }

};

and we'll make myVector2D inherit from this class:

class myVector2D : public myVector

{

public:

double getLengthSquared() { return x\*x + y\*y; }

double x, y;

};

Note that we didn't mention again that getLengthSquared() is virtual, the compiler will remember this. The definition of myVector3D, like before is:

class myVector3D : public myVector2D

{

public:

double getLengthSquared() { return x\*x + y\*y + z\*z; }

double z;

};

At this point we could define a function as follows:

bool isWithinUnitRadius(myVector &v)

{

if (v.getLengthSquared() < 1.0)

return true;

else

return false;

}

The beauty of this function, is that we can use any argument derived from the myVector class, and because the getLengthSquared function is virtual, the implementation belonging to that object will be used. So

myVector2D A;

myVector3D B;

isWithinUnitRadius(A);

isWithinUnitRadius(B);

is valid code, and in the first function myVector2D's implementation of getLengthSquared is used, while in the second function the myVector3D version is used.

Note that it is *very* important to use either a reference (&) or a pointer (\*) in the definition of isWithinUnitRadius. If the function would start like this:

bool isWithinUnitRadiusBAD(myVector v)

then a new myVector object would be created, based on the argument that was passed (which is still allowed to be a 2D or 3D vector). From that new object, then the getLengthSquared function is called, which is defined to always return zero. By using a reference or a pointer, the same object in memory will be used (so no copy will be made), and the virtual function will refer to that object's implementation.

Hopefully this simple example helps illustrate that virtual functions are a very powerful means of abstraction.

**Templates**

In C and C++, apart from using a double, you could also use a float. The first uses 64 bits to represent a floating point value while the second only uses 32. Needless to say, the 64 bit version is more precise, but if precision is not as important than memory for example, it's possible that a 2D vector using float values could be handy as well.

We could easily copy-paste the double precision version and create a new single precision version as follows:

class myVector2DSingle

{

public:

float getLengthSquared() { return x\*x + y\*y; }

float x, y;

};

Apart from the type used in the class definition, the return type of the function and the name of the class, we didn't change anything, so it would be really neat if we didn't have to copy-paste. This is where templates come in.

For this example, we could do the following:

template<typename T>

class genericVector2D

{

public:

T getLengthSquared() { return x\*x + y\*y; }

T x, y;

};

This tells the compiler that genericVector2D is not yet usable on its own but to finish it we need to specify a type for the template parameter T. To use this to declare variables, the following could be done:

genericVector2D<float> vecSingle;

genericVector2D<double> vecDouble;

In the way function parameters tell your program what to do at run time, template parameters tell your compiler how to finish a class definition or a function at compile time.

There are many very useful templates in C++'s *standard template library* (STL).

**Namespaces**

Suppose both Alice and Bob have created a myVector2D class, but Alice went for precision and used the double type, while Bob was more interested in memory and used the float version. If we'd like to use both versions in our own program, there's a fundamental problem since both classes are called myVector2D.

To help resolve such ambiguity, in C++ *namespaces* can be used. Using a namespace to prevent conflicts, Alice's implementation would be as follows

namespace alice

{

class myVector2D

{

public:

double getLengthSquared() { return x\*x + y\*y; }

double x, y;

};

}

while Bob's would be this:

namespace bob

{

class myVector2D

{

public:

float getLengthSquared() { return x\*x + y\*y; }

float x, y;

};

}

In our own code, we could specify a specific version using either the alice:: or bob:: prefix (in a similar way we did when defining member functions of a class)

alice::myVector2D v1;

bob::myVector2D v2;

If we're certain that in a specific source file we'll only be using Alice's implementation, we could write

using namespace alice;

and when mentioning just a myVector2D without prefix, the compiler will assume that's the version in the alice namespace should be used.

Standard C++ things are typically in the std namespace, including everything in the standard template library.

**The starting point**

A C or C++ program always starts by executing a special function called main. So when you don't really know where to start when looking at someone's code, looking for the main function is usually a good idea.

**Cyan**

To give you an idea about the code, I've put some things on-line at <http://research.edm.uhasselt.be/jori/simpact/> There, you'll find:

1. simpact-cyan-0.1.0.zip: A zip file containing the current source code. You can create an executable with it, but it's obviously not finished yet. It can be used on Linux and Windows platforms.
2. gsl\_win-1.15.zip: The GNU Scientific Library, compiled to be used on Windows. On Linux it will be either available by default or can be easily installed using your package management system.
3. simpact-vsc.webm: a screencast about how to create the executables on the VSC cluster (a Linux system)
4. simpact-VS2012.webm: a screencast about how to create the executables on a Windows platform, using the Visual Studio Express 2012 development environment. While you may need to register on some Microsoft site, this is a free piece of software.

About what's happening in the VSC screencast:

* First I download the zip file and extract it in some directory
* Building the executables will involve the creation of several other files and to keep the source directories clean, I'll be using  a separate directory called 'build' to build everything in
* The build system uses CMake, and the executables require GSL, so those modules are loaded on the VSC system
* Then 'cmake ..' is executed to generate the makefile which is actually used to build the executables
* 'make' is then executed to start the build.
* When the build is complete, four executables have been created: simpact-cyan-basic-debug, simpact-cyan-opt-debug, simpact-cyan-basic and simpact-cyan-opt. The 'debug' executables include lots of additional checks which the other executables don't use anymore. The 'basic' executables use a very simple version of the mNRM, in which the firing of an event causes all event times to be recalculated using the new simulation state. In the 'opt' version, a more advanced algorithm is used which tries to avoid unnecessary recalculation.
* Running one of the executables without arguments shows how to run the current version, and when parameters are provided, some output is generated. Currently, the only thing that's done is write to the screen which events are executed at which times

In the screencast for the Windows version, it's assumed that you've already installed Visual Studio Express 2012 (Windows Desktop Edition), which can be found here:<http://www.visualstudio.com/en-us/downloads#d-express-windows-desktop> You'll also need to have CMake installed, for which you can find a Windows installer at this page: <http://www.cmake.org/cmake/resources/software.html>

About the movie:

* First, the simpact and gsl zip files are downloaded.
* The GSL file is extracted in a way such that directories c:\local\bin, c:\local\include and c:\local\lib are created
* The simpact file is extracted in c:\projects
* The CMake GUI is started and is told where to find the sources and where to put the build directory
* To make sure that CMake can find GSL, we're adding CMAKE\_INSTALL\_PREFIX and setting it to c:/local. That way, it will search that location for the GSL files (sorry about the glitch in the movie there)
* Then, we press configure, selecting Visual Studio 11 (which is the version number of VS 2012)
* Then configure is pressed again, so that all that red stuff disappears, and finally 'Generate' to create the files needed for VS2012
* To build the executables, the 'Visual Studio Solution' file is double clicked, opening the VS2012 Integrated Development Environment (IDE)
* First, we'll build the 'Debug' versions of the executables.
* When this is done, the build type is changed to 'Release' and again ALL\_BUILD is selected to build the executables
* In the directory containing the Visual Studio Project file, you'll now find a 'Debug' and a 'Release' directory. In the Windows version, the executables names are the same for both versions but they're in different directories. What's in the Debug directory corresponds to the simpact-cyan-basic-debug and simpact-cyan-opt-debug versions from Linux.
* The executables have to be started from the command line, so a 'cmd' is launched in one of the directories in the file explorer
* But before we can run them, we need to tell the OS where to find the GSL DLL files.
* Basically, this is done by modifying the PATH environment variable (notice how I cleverly copy-paste a '\' there because my keyboard settings were incorrect ;) )
* This adjustment of the PATH only needs to be done once, it it saved in your system settings
* Then, the executables should work as in Linux.

The source code is in the 'src' subdirectory of the zip file, which itself is organized a bit further. Some highlights:

* The 'mnrm' subdir contains the mNRM framework as well as the implementation used in the 'basic' executables, in which everything is always recalculated
* The 'core' subdir contains everything for the more optimized, population based framework. This extends the code from 'mnrm'
* The code in the 'src' dir itself builds on the code in 'core', and provides (or will provide) the actual implementation for the simpact simulation

I've tried to separate things in a clean way, so that if we'd like to make another mNRM based simulation at some point, we can either start from the basic mNRM algorithm or a more population based algorithm, without necessarily having to start from the actual simpact implementation.