A selection of C++ core features and their proper use

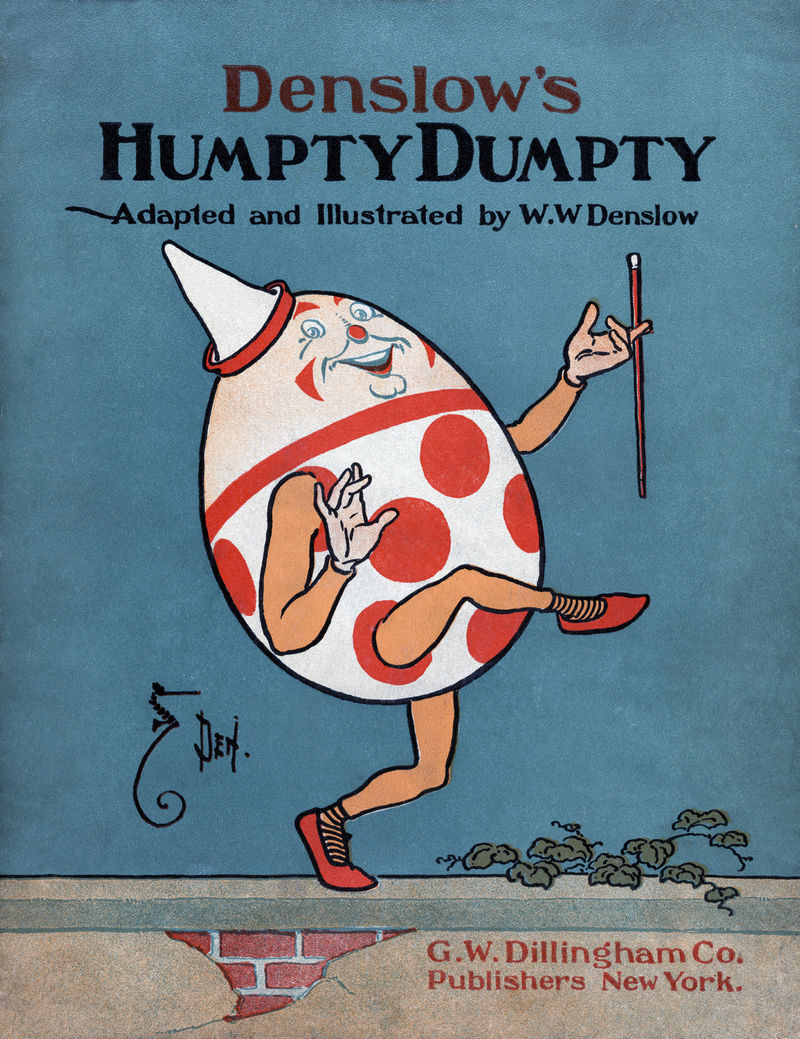
Without proper use of modules and classes, maintenance may cause software rot to penetrate your code: It used to work, I swear, but at some point it fell apart and "all the kings horses and all the kings man..."

J. de Hooge

September 2015

This text is freeware

(C) GEATEC engineering (www.geatec.com)



# Table of contents

[Table of contents 2](#_Toc430336056)

[Introduction 3](#_Toc430336057)

[What do I have to master and what determines my grade 6](#_Toc430336058)

[Getting started 8](#_Toc430336059)

[Using headers and includes to create a module hierarchy 9](#_Toc430336060)

[Pointers, memory management, arrays and references 13](#_Toc430336061)

[Pointers and memory management 13](#_Toc430336062)

[Arrays 14](#_Toc430336063)

[Character arrays 15](#_Toc430336064)

[References 17](#_Toc430336065)

[Object oriented programming in C++ 20](#_Toc430336066)

[Encapsulation 20](#_Toc430336067)

[Inheritance 21](#_Toc430336068)

[Polymorphism 22](#_Toc430336069)

[How to build durable and flexible object oriented software for technical applications 25](#_Toc430336070)

[Drawing up requirements 25](#_Toc430336071)

[Designing: a lightweight approach 26](#_Toc430336072)

[Coding 27](#_Toc430336073)

[Testing, reliability and simulation 28](#_Toc430336074)

[Constructor overloading and virtual destructors 30](#_Toc430336075)

[Templates 32](#_Toc430336076)

[Operator overloading 34](#_Toc430336077)

[A glance at some low level facilities 36](#_Toc430336078)

# Introduction

This document does not treat the whole of C++. Nor does the one semester course that it is part of teach you how to program. It's presupposed that you master at least one programming language, preferably an object oriented one. And it is also taken for granted that you can pickup rudimentary C++ from the countless tutorials on the Internet. Rather than replicate what is already abundantly available, this document tries to fill in the gaps, concentrating on stuff you really ought to know when programming in C++, but that's never the less seldom known to C++ novices. So start reading one of the many tutorials and try to make or at least understand the examples. After that pick up this document to find a highly practical, experience driven guideline to using this at times overwhelmingly feature rich language in practice. A camel is a horse designed by a committee. Well, after being conceived by Bjarne Stroustrup, C++ design was taken over by a committee, and a camel it is... But a very useful one for TI specialists.

*To really learn the essentials of C++ in a mere one semester course, attending the lessons is mandatory, even if you have some C++ experience. This text is only supportive to the oral explanation of this non-trivial stayer under the programming languages.*

C++ and its predecessor C are languages especially suitable for applications where predictable response time, explicit control over memory usage and direct hardware access are essential. These characteristics make C++ an indispensable tool for many technical applications.

Since C is by good approximation a subset of C++, in this document C++ will also refer to this subset. So if e.g. it is stated that C++ is often used for writing the operating system kernels this also implicitly includes kernels written in C.

With its predictability, efficiency and suitability for low level hardware access C++ is at the basis of almost every tool stack (compilers, interpreters, database management systems, shaders). As such it has largely replaced assembly language. Even when programming in a comfortable scripting language like Python, much of the work is actually done by libraries written in C++.

While C++ probably will be with us for a very long time, its age shows as "geological layers" in the language itself.

- The oldest layer is the plain C subset. While more sophisticated than the original Kernighan and Ritchie C language, C is still rather bare bones in modern eyes. It has no strings, no lists or any other dynamic data structures. Its infamous preprocessor much reminds of macro assemblers.

- The one but oldest one layer is the "classically object oriented" layer, featuring encapsulation (hiding implementation details), multiple inheritance (type extension c.q. specialization) and polymorphism (distinct object types, sharing a common inherited interface).

- A more recent layer implements generic typing, through the use of templates. This layer accounts for many subtle compiler differences and cryptic error reports, but also for a performance surpassing what is possible with polymorphism alone, sometimes at the cost of code bloat: object files (compiled code) become much larger than needed.

A suitably written C++ compiler is able to generate very compact code, measured in kB or even B. Sadly many compilers don't bother, generating MB's of code for simple programs. This sort of defeats the purpose of C++ and is in no way necessitated by the language itself.

As a consequence of the historical layers mentioned, C++ is not an easy language to learn. It is truly a multi-paradigm language (a language suitable for multiple, fundamentally different design styles). Many courses treat C++ as if it were a slight variation to the Java programming language. It is not. Although the syntax (outer looks) of Java is largely inspired on the syntax of C++, semantically (with respect to meaning) C++ and Java are very different.

Central to C++ is the pointer-concept and the explicit runtime allocation and deallocation of memory. It's this feature that gives C++ its power and efficiency. If you're not confident using pointers and dynamic memory, you have simply not learned C++ at all.

On the other hand there are many features in C++ that are certainly useful, but not essential for even production use of the language. Multiple inheritance and especially virtual inheritance are certainly the appropriate answer to some questions, but one can do without them at first. And while operator overloading shows the malleability of the language, and as such is useful to have a taste of, it does not add anything to the semantics and is prone to introducing confusion and inefficiency, e.g. by unnecessarily copying objects.

So for a first one semester course, choices should be and can be made. This document does make such choices. It does so as follows:

- Since there are many good quality tutorials on the Internet, many basic concepts and features are just mentioned: look up this and that, using search phrase such and so on Google. The ability to look up such things is indispensible for any software engineer. In many cases looking up things is left to your own initiative. If you can't follow the text of this document, look up the things you don't know or understand. So this document just deals with things you cannot easily look up. It's just the tip of the iceberg so to say.

- Less easy to answer by browsing the net is the question on how to *wisely use* certain features. A good example is the use of headers in such a way that, despite independent compilation and the lack of a real module mechanism, C++ applications can be structured as module hierarchies. Experience has shown that some ways to organize your code work better in practice than others. In such cases this document is somewhat dictatorial. It will often just state: Do this and that so and so without mentioning all the reasons why. As a student you're very welcome to differ of opinion and discuss with the author.

- Some features are important indeed, but are just too much for a one semester course. They are mentioned but not elaborated upon. If you take yourself seriously as a software designer, read about them and experiment with them if you can find the time. Not everything fits into the curriculum. If you want to develop interesting technical or scientific software rather than tailoring corporate administrative packages while peeking at the clock with one eye, you'll have to be able to absorb new knowledge all by yourself, during your entire working life.

- And then there are the back streets of the language. You'll walk into them at some point, but there's time enough to look up the details as you encounter them. Bit shifts and bitwise logic are indispensible when e.g. writing device drivers. But once you need such facilities, just lookup the syntax on the Internet, since the semantics are trivial.

- Lastly there are the dark corners of the language, like preprocessor trickery and plainly weird stuff like template metaprogramming (no need to look up unless you can't contain your curiosity). While occasionally you may find just what you need there, don't make it a habit to dwell in such places, since software rot may penetrate your code: It used to work, I swear, but at some point it magically fell apart and *"all the kings horses and all the kings men..."* (Google and front page of this document).

# What do I have to master and what determines my grade

To successfully conclude a first C++ course you'll have to be able to write C++ programs that:

* Do what they should do according to the requirements
* Use certain explicitly specified language features
* Are clearly understandable to humans
* Follow guidelines explicitly stated in this document
* Abide with good object oriented and modular design practice
* Have no memory leaks
* Can be explained per line and motivated per design choice by you

Language features of which you have to demonstrate mastery in your code are:

* Proper use of headers to enforce a hierarchical module structure.
* The control statements if..else, for, while, , do...while, break, continue, switch...case...default, return
* Pointers, pointer arithmetic, relation between arrays and pointers, allocation with new and deallocation with delete and delete [], stand alone references, reference parameters and reference returns.
* Stand alone functions.
* Classes and attributes, member functions, private and public parts (protected parts are useful etc.), default constructor, copy constructor and other constructors, function overloading, single inheritance, virtual functions.

Language features of which the use in the right spots is appreciated, but not mandated, are:

* Multiple inheritance
* Overloading of the \*, /, +, -, [], () and = operators
* Class and function templates

You'll have to be able to debug your program, either by printing intermediate results or by use of a source debugger.

You're grade is determined by the design quality of your code:

* A well designed program that has some minor flaws may still be rewarded with a sufficient grade
* A program that happens to work but lacks proper structure or clear naming conventions may not be considered sufficient

All assignments must be handed in within the specified time. A second chance is offered, being a completely new assignment.

Don't hesitate to ask for assistance. You won't be judged any lower for asking for help, on the contrary. Just copying code without being able to explain exactly what it does, however, will lead to unsuccessful termination of the course. The use in the assignments of other than the standard libraries a.o. STL is prohibited.

# Getting started

Install a compiler and an editor on your computer. Compiling from the command line is OK and actually preferable to finding out where essential options are hidden between all bells and whistles of an overly complex IDE at every new version. Contrary to popular wisdom a debugger is not needed at all, common sense and a few extra print statements will do quite as good.

Simple editors like Notepad++ (Windows) or Gedit (Linux) will do fine, especially in combination with a small compilation script. Don't waste energy on make files or complex IDE's.

Look for C++ tutorials (Google) on the internet. Program our own version of "Hello World". After that look at classes, inheritance and polymorphism (Google for "virtual function"). Make trivial programs to try out these concepts. After that you're set to go!

# Using headers and includes to create a module hierarchy

C++ features independent compilation. This means that a C++ compiler sees only one source file (.cpp file) at a time.

Suppose you define a function in file *definition.cpp* as follows

*void greet (int i) {*

*cout << "Hello, Rotterdam " << i << endl ()*

*}*

and you use it in file *invocation.cpp* as follows:

*if (a > b) {*

*greet (3.5)*

*}*

then the compiler is not able to detect that you define the function with an integer parameter and call it with a float, since it will never see the two files in combination, just one by one. Eventual the two compiled files *are* connected together by the linker. This is a separate step after compilation. Linkers, however, are allowed to be as dumb as a potato. They don't know about types, just about memory addresses. So your program will compile ok, will link ok but will print gibberish rather than 3.5, since it tries to interpret something stored in memory as a float as if it were and integer.

To solve this, C++ uses the so called linking pin principle. A linking pin in an organization is a person that is a member of two otherwise disjoint groups of people. Suppose you run a newspaper and there's a text department and a pictures department. In order to get the right pictures with the right text, it's quite handy to have at least one person that is a member of both departments. This person is called the linking pin.

In C++ the linking pin is the header file. To achieve cross module type checking, use headers in the following way:

* Put all your class declarations in a header , e.g. module1.h, including *function prototypes* (Google, if you don't know what those are).
* Put all your function implementations, i.e. prototype + *function body* (Google), in a cpp file with the same name before the dot, e.g. module1.cpp
* *#include* (Google) module1.h in module1.cpp
* #include module1.h in any other module (including the main module) that uses module1.cpp

Build up your modules hierarchically, so try to think in levels c.q. layers. Performing disk I/O is at a lower level than interpreting SQL. Utilizing SQL to archive data on your music collection is even at a higher level.

As can be seen in the diagram above, module File\_access is used in module SQL\_interpreter, as well as in the main module Music\_collection and in module Audio\_access. Never have an arrow running from a high level module to a low level module. So never use a high level module in a low level module. This would make your lower, more general level dependent on your higher, more specific level, which would make it impossible to reuse your lower, more general level modules in a different high level context. If e.g. module File\_access would depend on module SQL\_interpreter (red dashed line), you would not be able to implement a non-relational (non SQL) database without having all your SQL stuff around. Having arrows running from high level to low level modules gives you spaghetti by design. Try to pull one module out with a fork and all the other ones cling to it like big lump of sticky pasta. This type of spaghetti is not to be confused with the famous goto spaghetti, *"considered harmful"* ( Google) by some, but at least as bad.

Figure 1 Example of a module hierarchy with three layers

Jukebox.cpp

SQL\_interpreter.h

SQL\_interpreter.cppp

MP4\_codec.h

MP4\_codec.cpp

File\_access.h

File\_access.cpp

Audio\_access.h

Audio\_access.cpp

= included in

= inclusion prohibited

bottom layer

middle layer

top layer

In computer program design there are no dogma's. So in due time, having gained enough experience, everything is open to reconsideration. In the HR C++ course, however, stick to the following guidelines unless you have a very strong reason to deviate from them:

* Start with an empty file, so remove any code autogenerated by the IDE for new projects, like references to *stdafx.h* and an alternative main function, or generate your projects not to include such manufacturer dependent code noise in the first place.
* Start every header with *# pragma once*. This prevents indirect multiple inclusion of headers, which may lead to slow compilation at best but may also yield errors.
* Define (or rather, to use the appropriate slang, declare) only classes in your header file, so no standalone variables c.q. objects or functions. In these classes there will usually be variables and declarations of function prototypes.
* Define functions including their body only in the C++ file, unless you're using templates. In the latter case it is acceptable to define them inline (inside your class). Note that this will in many cases lead to code bloat (large object files and executables), since code for the same function will be generated at every inclusion of the header.
* Make everything in your class that should be accessible from outside *public*. Make the rest *private*. For the time being forget about *protected*.
* Don't use getters and setters for every attribute in your class. While with Java this facilitates the construction of configurable "beans", in C++ it just introduces lots of unnecessary code and a severe performance drop. Getters and setters can be very useful from a design point of view, since they enable actions triggered by data changes. Rather than blindly applying this everywhere as in Java, just use them if you need them. If speed and code size do not matter, then why choose C++ in the first place.
* Use clear, *camel case* (Google) variable names. Classes start with a capital, everything else doesn't, despite traditions of a certain Redmond based software manufacturer. Constants are often typed in uppercase. This tradition is not very useful anymore, since constants are no longer macro's but just variables "that do not often change", but it's allowed, though not advocated, and unfortunately probably expected by your future colleagues.
* There's no need to use NULL for a zero pointer, 0 is just as clear. In general don't code type information in your literals or variable names unless you have special reasons to. Names like cpciCounter (constant pointer to constant integer counter) are a thing of the past, given the explosion of possible types now that you can define your own classes. Leave type checking to the compiler and use your names solely to indicate what a variable is used for.
* Add comments only where needed, but if needed, do add them indeed. A statement like

*i++; // Increment the record counter*

should be

*recordCounter++;*

* The practice of *"literate programming"* (Google) is appreciated. Note that this is radically different from adding large quantities of trivial comments which just add to the amount of text that a reader of your code has to wade through, without adding any clarity or insight.
* Think before you program! It is completely OK to think in a programming language, but if you do so, work in a structured way, e.g. from crude to fine rather than in statement order. Declaring the interfaces of your classes first is often a good way to start. Another way that works very well in practice, is to use objects as if classes were already defined to establish usage patterns and after that actually define your classes, since now you know what they must be able to do. Use UML diagrams if they help you getting your mind around things, especially the class diagram and the sequence diagram may be useful, but don't indulge in UML formalities.

# Pointers, memory management, arrays and references

This paragraph is complementary to plethora of knowledge on these subjects available on the Internet. Start out by investing an hour reading about these facilities (Google). Try to make some simple example programs to test your understanding. Having this as a basis, mind the following.

## Pointers and memory management

A pointer is a variable that holds the address of another variable. Note that an object of a certain class is just a variable of a certain, often self defined, type, in other words don't be intimidated by the object orientation slang. So pointers can hold the addresses of objects as well, since these are just a special type of variables. Some even say that all variables in C++ are objects. While this is true for e.g. a language like Python, it is at best not common terminology in C++. Still we'll use the terms variable and object interchangeably in this text.

So suppose we have a variable defined as:

*int i = 3;*

Then a pointer to this variable would be:

*int \*pI = &i;*

This takes some getting used to. Read it as follows: If I dereference *pI*, so take the variable that pI points to, I'll find an integer, in this case *i*.

Put the star adjacent to *pI* rather than *int*, so indeed *int \*pI*, NOT *int\* pI*. The reason for this mandatory convention becomes clear if we try to define two pointers in one line of code:

*int \*pI = &i, \*pJ = &j;*

As you can see the *\** has to be repeated, it's not part of the type but rather a dereferencing prefix to the variable, dereferencing meaning finding what the pointer points to. In line with this *\** is called the "dereference operator", and & is the address operator , taking the address of a variable.

The nice thing about pointers is that they can point to things that are not on the *stack* but on the *heap*. Let's see what those terms mean. Normally, variables are defined inside functions. Such a variable is placed on top of a pile of previously defined variables in memory. That pile is called the stack. As soon as the function is left, the stack space for the variable is released again. So the variable that was added as last one is deleted first (LIFO, Last In First Out). Such a ordinary variable is called an "automatic" variable, since it is automatically released as its *scope* (Google) ends. But sometimes we want variables to survive when the function that created them exits, e.g. when they form a complex datastructure like a *linked list* (Google) or a  *tree* (Google + *datastructure*). Such survivers are not *allocated* (Google) on the stack, but from the *heap*, an at first sight rather chaotic parking lot of variables that can be put in or taken out randomly. Putting them in happens by means of the *new* statement, e.g. as follows:

*int \*pI = new int;*

Since such a so called "dynamically allocated" or "dynamic" variable is not automatically released, the programmer has to do so if the variable is no longer needed:

*delete pI;*

Note that this does not delete pointer *pI*, but rather the anonymous integer that it points to and that was allocated with *new*. Dealing efficiently with heap memory is kind of an art, after a lot of allocating and deallocating it the heap starts to look like an Edammer Cheese, or rather indeed like a parking lot where many small cars have left, leaving places open that are too small for the bigger ones. This is called memory fragmentation. Failing to call *delete* variables altogether leads to a so called *memory leak*: eventually the parking lot gets filled up with cars that should have gone to the junkyard. Especially if the pointer itself, so not the thing it points to, is released after leaving the function that defined it, the address of the piece of memory that it pointed to may be forever lost until the application exits. It's like the owner of the car died without ever collecting it from the parking lot. Since operating systems in themselves may leak as well, a worst case scenario well known to PC users (no Bill, not talking about you) is that memory will only be reclaimed after a cold boot.

Given the current amounts of RAM memory this doesn't seem a large problem, unless careless programmers forget to deallocate complete video movie objects, as in fact they do.

Since you'll be designing technical systems, sometimes operating on single chip with a possibly tiny memory footprint, avoiding memory leaks is a must in this course. And: it's just civilized as throwing trash in a dustbin.

In general it's advisable to allocate variables only in a *constructor* (Google) and to deallocate them only in a *destructor* (Google). This simple strategy will effectively prevent all leaks, but is sometimes impractical. Still: adhere to it in this course.

Sounds primitive? Well, it is a matter of precise control as is needed in critical technical applications. The alternative is *automatic garbage collection* (Google), which has by nature unpredictable real time behavior, inefficient memory use and administrative overhead. There exist automatic garbage collection libraries for C++. They may incidentally be useful, but if you have to use them a lot, consider combining C++ with e.g. *Python or Cython* (Google), a combination that has become immensely popular for technical and scientific computing.

## Arrays

C++ natively does support *arrays* (Google). Arrays are single dimensional, but can be nested to give you *jagged arrays* (Google).

An array is declared e.g. as follows:

*float floatArray [10];*

THE NAME OF AN ARRAY IS A POINTER TO ITS FIRST ELEMENT.

So:

*floatArray == &(floatArray [0]);*

(For the insiders: no energy is devoted to even attempting to explain the C++ *const* disaster in this text.)

Now suppose we have a pointer that points to the first element of an array. If the pointer is incremented by one, it will point to the next element, no matter how big or small these elements are. So if a pointer is incremented by one, but it points to a 64MB object, it will point to the next 64MB object. A fortunate consequence of this is the following equivalence:

*anArray [i] == \*(anArray + i);*

*Understanding this equivalence is an absolute necessity for a C++ programmer* (as opposed to a Java programmer that has dwelled across the border of alien territory). It's perfectly alright if you don't dig this at first sight, but the ASK THE TEACHER TO EXPLAIN!!! You know he actually gets *paid* to do this...

Something special happens if arrays are allocated from the heap:

*float \* floatArray = new float [10]; // With an array, "new" stores size information*

*// ... use it*

*delete [] floatArray; // The [] causes the size info to be used to delete this array*

## Character arrays

A special role in C++ is played by character arrays. A character array is just that, an array of characters. It has no special facilities like information about its length or where it ends.

*char charArray [10];*

*char [3] = 'x'; // All other elements still undefined.*

The special thing about char arrays is the way they can be initialized:

*char \*charArray2 = "Mary"; // charArray is, as usual,a pointer to the first element of the array*

In this simple statement, several things happen:

- Memory space for 5 characters is allocated, from the stack or globally.

- The first 4 locations are filled with the characters 'M', 'a', 'r' and 'y' respectively.

- The 5th location is filled with char (0), a character with code 0, denoting the end of the array.

- Such a character array terminated by a 0 character (char (0), NOT '0') is traditionally called a "zero" terminated string.

So C++ itself doesn't have a separated string type. Strings are just character arrays that happen to have char (0) at the end. No use attempting something like

*charArray2 ="Isobel";*

since variable *charArray2*  has only length 5, whereas string literal "Isobel" takes 7 characters (Why not 6?).

So how about:

*char \*charArray3 = "John";*

*charArray2 = charArray3;*

That would fit, wouldn't it?

Yes it would, but no characters are copied. Pointer *charArray3* is just assigned to pointer *charArray2*, making them both point to the same memory location, which gives us two names for the same zero terminated string.

To really copy a zero terminated string:

- Allocate memory for the new string:

*char charArray2 [5];*

- Copy the characters by calling a standard function called *strcpy*:

*strcpy (charArray2, charArray3);*

Wow, that's primitive!

Let's get even more primitive, by exploring how *strcpy* actually works on the inside:

*void strcpy (char \*pDestination, char \*pSource) {*

*while (\*pDestination++ = \*pSource++);*

*}*

Note that an assignment in itself is an expression, and that *char (0)* is interpreted as *false*.

Like it or not, to call yourself a C++ programmer, you'll have to fully and exactly understand how the *code of strcpy* (Google, your teacher) works. This terse source code leads to generation of very tight machine code. Assembly language could not have been more efficient. This is all part of C++. You can manipulate individual bits, express your preference to put variables in processor registers rather than in RAM memory etc.. C++ is standing with its feet in the mud of the hardware.

*Remark: It is sometimes stated that humans shouldn't be bothered with peephole optimizing tasks like this. Compiler optimizers should turn unwieldy source code into the tightest machine code. Yes, in an ideal world people would state their algorithm in plain English and the compiler would take over from there. However we're not quite there yet. Awareness of what machine code is generated certainly helps thinking in terms of efficiency and, as a byproduct of that, also paves the way for conciseness. Not paying any attention to these matters is like learning arithmetic by using a pocket calculator.*

Still don't like mud? Don't worry, you're not obliged to wade through it. C++ comes with the STL (Standard Template Library) that has got all the luxurious data structures one would need: *string, set, map, vector* etc.. In the sequel we'll use *string*rather than *char\**. But if you're ever tight on memory or processor time, programming a 4 bit coffee machine control, remember the mud on which the lotus flower grows.

## References

In C, function parameters could only be passed by value. This means that the value of parameters are copied as automatic variables (explained earlier) on the stack, to be available inside the function. Since the variable is released at exit of the function, it cannot be used to pass information back to the caller.

To this end C programmers traditionally use pointers which results in a lot of spooky characters (&, \*) in your code.

*void increment (int \*pI) {*

*(\*pI)++; // Braces added for clarity*

*}*

*int a = 3;*

*increment (&a);*

*cout << a; // Prints 4*

Alternatively you can use references. A reference is denoted by &. This is confusing, since & is also the address operator, but it will be explained soon how you can see the difference.

The same functionality is now achieved by:

*void increment (int &i) {*

*i++;*

*}*

*int a = 3;*

*increment (a);*

*cout << a; // Prints 4*

As you can see the version with references has less spooky characters, so is more readable to most. Effectively it provides C++ with "pass by reference", as in e.g. *Pascal* (Google, but only if you're interested in ancient history). Since in the second function the effect of (*\*pI)++* is achieved simply by writing *i++* where *i* is a reference, it is said a reference is a "pointer with a built-in \* prefix": references are simply dereferenced pointers. This is illustrated by the fact that the body of the function can be compiled to exactly the same machine code as in the reference case, only the notation is simpler if references are used. Contrary to pointers, references cannot be made to reference anything else than the variable they originally "point to".

But references can do some other useful tricks:

*int i = 3;*

*int &j = i; // j becomes a synonym for i, so i and j actually denote the same variable*

*j++;*

*cout << i; // prints 4*

This can be handy in many cases:

*Car &car = country.state.town.street.car;*

*car = Car ("Volkswagen") ; // Changes the content of country.state.town.car*

Also, functions can return references, allowing function calls to the left of the = operator:

*float &lowestGrade (float &x, float & y) {*

*return x < y ? x : y; // Google "conditional expression"*

*/\**

*Or, more familiar but also more verbose:*

*if (x < y) { // Use braces to be sure, prevents mistakes when adding statements*

*return x;*

*}*

*else {*

*return y;*

*}*

*\*/*

*}*

*grade1 = 4;*

*grade2 = 6;*

*lowestGrade (grade1, grade2) += 2; // grade1 will have become 6 after this*

But how to you distinguish *&* meaning reference from *&* meaning address operator? There's a simple rule for this:

IF AND ONLY IF *&* IS PRECEDED BY A TYPE, IT'S A REFERENCE,

IF AND ONLY IF *&* IS PRECEDED BY ANYTHING ELSE, IT'S AN ADDRESS OPERATOR.

If you don't clearly understand this rule, ask your teacher!

Experiment with pointers, address operators, references and arrays, new and delete. Dealing with memory in this direct way is what makes C++ suitable for applications that are close to the hardware, like operating systems, drivers and microcontrollers. *You'll get the hang of it if you use it frequently in small programming experiments, not by merely reading or listening!*

# Object oriented programming in C++

Three language facilities in C++ make it possible to adhere to a programming style called object oriented programming. Thos facilities are *encapsulation, inheritance and polymorphism*. In the sequel these are treated one by one.

## Encapsulation

Encapsulation means that lumps of data and the functions that operate upon them are brought together in a common housing called an *object*. Objects are instances of classes, just as variables are occurences of types. We may have a class Dog (classes are started with a capital letter), but that does not mean we have any dogs around. There still exists a class TRex, however there do not exists any living instances of this class anymore.

Since tRexes are all deallocated, lets continue with the dogs:

The class declaration will usually reside in the header file, in this case e.g. animals.h

*class Dog {*

*public:*

*Dog (string name); // Constructor*

*void makeSound (); // Method*

*void tellName () ; // Method*

*private:*

*string name; // "field' or "attribute"*

*}; // This semicolon is often forgotten, causingan avalanche of errors*

The methods are usually elaborated in the corresponding C++ file, in this case e.g. animals.cpp

*Dog::Dog (string name ): name (name) { // Initialize field "name"*

*}*

*Dog:: makeSound () (*

*cout << "wraff!" << endl;*

*}*

*Dog::tellName () {*

*cout << "My name is " << name << endl;*

*}*

Now, in the main function of the program, let's create a dog and bring it to life:

*Dog dog ("Cujo"); // Or, alternatively, Dog dog = "Cujo"*

*dog.makeSound ();*

*dog.tellName ();*

Lets create another one, this time from the heap:

*Dog \*dog2 = new Dog ("Fluffy");*

*dog2.makeSound ();*

*dog2.tellName ();*

*delete dog2 ();*

Note that the dog's name is stored in the so-called private part of the object. The label *public* means anyone can use that particular method or field. The label *private* means only methods of the same class can use it. There's a third label: *protected* (Google). We don't use it in this course. Note that *public*, *private* and *protected* are valid for a group of methods and fields that follow, not just for one method or field as is the case in e.g. Java.

It is said that especially using the label *private* is what encapsulation is about: protecting things from unauthorized access. This is a very defensive view of encapsulation, easily leading to hoards of unneeded getters and setters as in Java. Encapsulation is primarily packing methods and fields in a common capsule: the object. Whether or not access to parts of such an object is limited is a secondary matter. There are some very successful l languages relying entirely on hints and good will with regard to what is accessible and what not. In the context of a cruise missile control such a "high trust" strategy may not be the right approach. But on the other hand, simplicity is probably the single largest contributor to reliability.

## Inheritance

Inheritance is building one class as extension c.q. enrichment c.q. special case of another, adding methods and fields to the ancestor class. Since it is indeed the implementation code that is inherited, such inheritance is often called implementation inheritance. In the upswing of object orientation it was said that implementation inheritance was *the absolute and groundbreaking silver bullet* or this then new programming paradigm (Google), enabling code reuse and saving lots of time, money and effort. This has turned out to be almost completely false. Design flaws are as inheritable as features and while a very carefully designed class hierarchy may stand through storms of requirement changes, in practice many existing class hierarchies designed in-company as mere part of an application are a hindrance rather than an asset. The following situation is typical: A company started out with an adequate class hierarchy some 10 to 20 years ago. It was very well thought through, but unfortunately the original designer left after 5 years or so. From that point on, people have been gluing, copy-pasting and finger-crossing their way through the source code. What's left is a heavily interconnected, totally incomprehensible, very object oriented mess. The author of this text has seen companies struggling with such a heritage for 10 years, a struggle they eventually lost. Sometimes it's important to throw something away and start all over again.

INHERITANCE IS NOT MAINLY ABOUT CODE REUSE,

although some beneficial reuse may actually happen,

INHERITANCE IS MAINLY ABOUT INTERFACES.

If there's any silver bullet in programming, it is designing lean and stable interfaces. Interfaces determine how the parts of a program cooperate. The purpose of in interface is to insulate the details of parts of your program from each other.

Suppose you have a program consisting of k parts and, simplistically, in each part there are n detailed design decisions to be made. And suppose each design decision has the potential to influence the possible decisions in all other modules is n x n x ... x n = kn.

Now suppose each module has a stable interface, that does not depend upon the outcome of any detailed design decision, but takes an abstract, high level view. Since now the n detailed design decisions of one module don't influence the possible decisions in any other module the number of possible design decision outcomes that have to be considered is k x n.

So by using stable interfaces we go from exponential complexity to linear complexity, as a function of program size. Exponential complexity means that adding any feature gets more and more complex as an application grows. Linear complexity means that adding the 100th feature is not fundamentally harder than adding the first one.

While this is only a thought experiment, the effect of large, time grown applications to become as flexible as a petrified dinosaur, is well known in practice. This is exponential complexity doing its regrettable job.

In practice there's usually a point in time where an existing design should be thoroughly reorganized if not scrapped. With a well designed set of classes and modules featuring interfaces that are stable, i.e. resilient against internal changes in a module, this point is reached much later. Such code is deemed flexible.

Especially in technical applications, that easily have a lifetime of "20 years +" in a hardware environment that regularly changes due to technological progress, software flexibility counts.

## Polymorphism

Polymorphism is the storage of objects of different types, but with a common interface, together in one data structure.

// ====== animals.h

*class Animal { // Abstract base class, since it has at least one pure virtual function*

*public:*

*virtual void makeSound () = 0; // Pure virtual function*

*};*

*class Dog: public Animal {*

*public:*

*virtual void makeSound ();*

*};*

*class Cat: public Animal {*

*public:*

*virtual void makeSound ();*

*};*

*// ====== animals.cpp*

*#include <stdio.h>*

*#include "animals.h" // Module header as linking pin*

*virtual void Dog::makeSound () {*

*cout << "Wrraf";*

*}*

*virtual void Cat:: makeSound () {*

*cout << "Mrrauw";*

*}*

*// ====== zoo.cpp*

*#include "animals.h" // Module header as linking pin*

*Animal \*animals [10];*

*for (int i = 0; i < 10; i++) {*

*if (i < 5) {*

*animals [i] = new Dog ();*

*else {*

*animls [i] = new Cat ();*

*}*

*}*

*for (int i = 0; i < 10; i++) {*

*animal [i] -> makeSound (); // Means the same as (\*(animal [i])).makeSound ()*

*}*

A polymorphic datastructure is a datastructure containing pointers or references of type pointer to ancestor or reference to ancestor respectively. But these pointers or references in fact point to objects of diverse derived classes. So pointers or references to objects of distinct classes can be held in the same datastructure, as long as they have a common base class. Array *animals* in file *zoo.cpp* of the example is a polymorphic datastructure, being of type "array of pointer to Animal", but actually containing pointers to Cats and Dogs.

A *virtual* function is a function that is selected via the class of actual object, rather than via the class of the pointer or reference to that object. In the example, the functions *makeSound* of Cat or Dog are called, rather than the *makeSound* of Animal. This would also have been the case if *makeSound* of Animal would have had a body. If on the other hand a function is non-virtual, the function belonging to the class of the pointer or reference is called, not the one belonging to the class of the object of a derived class that the pointer points to. In the latter, non-virtual case, of course, the function must have a body, otherwise it cannot be called.

A pure virtual function, sometimes mystically called a pure function, is a virtual function without a body. It cannot be called and is a mere placeholder to define an interface.

An abstract class is a class that contains one or more pure virtual functions. I's descendants have to provide an implementation (a body) for these functions. If a class inherits from an abstract base class, but does not provide bodies for all its pure virtual functions, it is still an abstract class. If in a class at any level of an inheritance hierarchy no pure virtual functions remain, i.e. each virtual function has been provided with a body, this class is not abstract anymore. We could call it concrete, but the use of this term is not widespread.

Note the elegant simplicity and regularity of this mechanism. Rather than explicitly labeling a class as an (abstract) interface as in e.g. Java, in C++, the compiler takes care of finding out what is abstract (due to the presence of at least one pure virtual function) and what is not. Since C++ features multiple inheritance (inheriting from multiple classes), any class may take the role of defining an interface for its heirs.

However, classes that define an interface can also contain implementation code. So there's no need to implement an interface over and over again in a class hierarchy.

Remark: Multiple inheritance opens the possibility that a field is inherited via multiple parent classes. In order to determine whether the descendant class contains one or rather more instances of that field, the virtual inheritance facility is part of C++. Virtual inheritance is outside the scope of a first course in C++ and hence of this text.

# How to build durable and flexible object oriented software for technical applications

This chapter is not about C++ in itself, but about issues that are intimately connected with programming technical applications in C++ with use of the language facilities treated up to now. While you can probably get a sufficient grade for C++ whilst skipping this chapter, if you plan to make development of technical applications your job, it is a useful read.

## Drawing up requirements

Building any type of software starts with drawing up requirements. Neither voluminous documents, nor dogmatic use of diagrams from whatever method is popular at the moment will in itself yield effective requirements. Good requirements are brief yet concise. There's much truth in the saying that the success of a project is determined in the requirements phase. For any serious technical or scientific application, specify the absolute bare minimum of what an application has been able to do to satisfy practical needs. In many cases it will turn out that the result is a robust system that is just feasible and just affordable.

It is absolutely necessary to weed out all bells and whistles before the design phase. Too many systems are born lifeless as a result of requiring unnecessary features. Such features can be added to a well designed, i.e. flexible, kernel in a later phase.

Be careful when estimating effort. Again as a practical rule of the thumb: Try to seriously estimate how much time it will cost to build the system and multiply that by three (some say pi). It will in most cases turn out that it's possible to remain within planning and budget without significant margin if a skilled team of developers is at hand.

Especially for object oriented software it is important that the terminology used in the requirements stems from the application area, rather than from the IT world. Different from IT terminology, application area terminology tends to stay the same over many years. An oil- and gas surveying application is about boreholes, wells, lithology types and seismic data, not about trees, lists and hashtables. A requirements specification should contain a brief domain description, i.e. explanation of the (a.o. physical) context in which the system has to function.

Effective requirements should be brief and mention essentials first. Contrary to what one might expect, requirements for multimillion dollar projects often do not land at all in the minds of relevant stakeholders in the project. Many stakeholders only start to really think about a system once its realization is well under way. Simulation in the requirements phase is a very powerful way to encourage an early start of this thinking process. Keeping requirements brief and making them an attractive read is another way.

What does not work well , is granting the requirements a rigid juridical status and refusing to meet the customer's needs because "that is not what we agreed upon". The customer cannot accept a system that doesn't work in practice, and will either find a way to force the fulfillment of his needs or bail out of the contract and terminate business with your company. In both cases time and energy is wasted that should have been invested in solutions rather than quarreling.

On the other hand it is important to honestly and firmly limit the requirements to what is feasible. A customer will in general accept sound technical arguments. If not, shore turns ship and it's better to terminate the project as early as possible.

## Designing: a lightweight approach

Experienced C++ programmers think in code. This is an advantage, not a limitation. Setting up a class hierarchy by indeed coding the public part of class declarations in experienced hands is a quick and concise way to lay down the backbone of a design. Working in this way, and not e.g. starting out with UML diagrams can be very effective. If diagrams are needed, keep them as informal and simple as possible. Although it is always possible to generate class diagrams from C++ code, the benefits of this approach in the design phase are marginal at best.

Especially technical systems can be long lived (30 years +), and will undergo modifications as e.g. hardware, safety guidelines and the production process changes. These modifications are often not done by the original developers and will in most cases rely on the sourcecode rather than on any documentation. Possibly at that point class diagrams and call graphs *will* be generated retrospectively from the original source, to aid reverse engineering. Voluminous, formalistic, overly detailed design descriptions and diagrams will be largely ignored, should they still be available.

So your design should be in your code! For technical software, the method of Literate Programming (Google) will usually lead to better results. Design (rather than reverse engineering) is not about duplicating code in diagrams or vice versa, to be useful it should apply to a higher abstraction level than the code itself. By starting with brief, concise comment blocks describing the overall modules and their usage design and code are merged in an application that is both readable to humans and to the compiler. This approach can be expanded to class and even function level.

An example of this approach on function level in a C++ header is given below:

*void adminGet (TLocator completeSourceLocator, TContainerId containerId, bool predicted = false);*

*/\**

*Actions:*

*If source locator known*

*If source locator occupied in stackimage*

*If containerId parameter differs from containerId in stackimage*

*Report error*

*Clear source locator*

*Else*

*Report error*

*Call correctStackimage with parameter completeSourceLocator*

*Else*

*Report error*

*Assign containerId parameter to spreader locator (P)*

*Check uniqueness of id of picked up container*

*If not predicted*

*Call predictStackImage*

*Remarks*

*Error process will asynchroneously prompt crane driver for eventual unknown id's, spreader locator first*

*\*/*

The above brief description was put in the header before writing any code. The actual implementation of this function is just an elaboration of this description, but is about five pages long and not easy to grab without first reading the description. When putting descriptions and design considerations in briefly in the code like this, there's an optimal chance that they get updated if the code is modified.

Whereas descriptions on class and function level are useful to guide coding as well as reverse engineering, descriptions on module level facilitate the division of work over multiple developers and modular testing. Module tests (unit tests) are to be written at the same time that a module is written, and are used right from the beginning but also as regression test in case of modifications after commissioning (Google). A good module description is at the same time a specification of a module's interface to other modules, of its functioning as a black box, and of its testable behavior.

Designing C++ applications, in fact all object oriented applications, works from two sides. After having specified requirements, you can work top-down, describing your modules and their interfaces (after this the tasks can be effectively split among developers), then specifying your classes, especially the public part, then describing the main functions, and after that starting to code.

But there's also activity in the opposite, bottom-up, direction. Certain low level facilities can be designed and build even if the top-down design is not ready. This holds for general facilities like communication protocols, realtime control loops and GUI elements. But it also holds for predictable bottlenecks, like facilities that are bound to be either time critical or memory hungry. Tackling these bottlenecks early diminishes the breakoff risk. Last but not least, designing partially bottom-up enlarges the possibilities for code (component) reuse, as a part is designed without relying on the specific context of the level immediately above it.

## Coding

Since a good design entertains a certain degree of abstraction, it will rarely be worked out on statement level. For non-trivial technical applications there's no strict border between designing and coding. Coding isn't the work of a typist "translating" a design, but of a skilled developer taking the design to the detail level.

When referring to coding style, programmers often mean the way indentation is used, formalistic naming conventions, boilerplate comments that add no value, but HAVE to be present, getters and setters for each field, etc. These syntactical and typographic issues however, are far less relevant than the semantics (Google). A strong focus on such matters of form rather than function is an indication that more important design considerations are underexposed.

It is not typography that makes a program comprehensible, but the fact that its developer had a clear, thorough understanding of the problem at hand. Microsoft used to have a complicated system of prefixes to (completely redundantly) encode the type of variables (Google for Hungarian Notation). They wisely left that track and stopped shoehorning young, talented developers into a least common denominator system. This moment coincided with a dramatic rise in the code quality of their system software.

Some conventions in C++ coding are widespread and useful:

- Class names start with a capital

- Segmented words use camel case: MyVerySpecialClass, myNotSoSpecialObjectOrVariable, anyFunctionExceptWhenItIsPartOfDotNet ()

Not so widespread, achieving a better on-screen overview by not wasting lines:

- Open curly brackets at the end of the previous line rather than at the start of the next.

- Putting each code fragment controlled by an if, while or for between curly braces, to prevent misleading indentation and forgotten braces if statements get added

## Testing, reliability and simulation

As a result of the quest for raw speed and low memory consumption, C++ has no built-in protection from errors like dangling pointers (Google) and buffer overflows (Google). A sound understanding of C++ pointers, references, arrays, dynamic memory management and the appropriate use of constructors and destructors are the first and most important line of defense here. Using libraries that do have this protection is another possibility if there's enough memory or processor time.

If you deed a debugger to find out what's wrong with any of the programs resulting from the assignments in an introductory C++ course, you're getting it backward. Your mental model of what you are making should be sufficiently clear to avoid such errors right from the start.

Still, testing is needed to spot thinking errors, especially missing certain "use cases" (Google). For a module the use cases should be explicitly in the module description. A module test harness (Google) should be written at the same time the module is written, or in advance. In general, testing code should be added early, and be left available (although probably deactivated) for regression tests (Google).

It is quite common that individual developers contribute modules to a larger software system that contain errors that would have been found with any even minimal amount of systematic testing. This is wicked. As a technical programmer you are a craftsman, whatever your manager says, and it is a matter of personal morale to try and deliver bug free code, especially in a technical system where bugs can result in injuries and loss of life.

Nevertheless: You'll miss things, despite all efforts. And if you don't, the operating system on top of which your software runs isn't perfect. Yes, that also holds for Linux. And the libraries you use aren't perfect as well. So in technical systems there should always be a hardware safety level. The rising flood of partially or wholy computer controlled cars is likely to make the relevance of this known to a wider audience, but you'd better be ahead of the pack on this one. In the extreme, being confronted with the question whether you even *want* to make this or that control program is a very real possibility, that the author has met repeatedly in his working life. Sometimes you will be the only one that can properly estimate the risks. In that case it is important to take your responsibility. Telling yourself your manager was responsible will not make you sleep well after a serious accident.

Apart from testing, desk checking will enhance reliability and safety. Read through your (compact, readable) source together with a colleague. By explaining, it'll turn out you spot our own thinking errors more easily. In a C++ course, it's good to team up with a fellow student. You may expand this approach to "pair programming" (Google). But you'll also need time to think, so a mix of pair and solo programming is advisable.

And then there is simulation. It is quite possible to simulate both a (number of) controls and a (number of) controlled system(s) written in C++ in a reliable way. Following this road will dramatically shorten commissioning time and at the same time increase reliability by orders of magnitude. Note that simulation will actually save lots of time, not cost any: It's again quite normal for a system to be debugged on the production location, controlling the actual hardware. As all kinds of activities are being performed there, and participants often are under contractual pressure, blaming each other for problems, this will typically costs months. With simulation this can be reduced to days and a far better system will result. Exaggerated, you think? Remember the two most recent attractions of our national amusement park, a pirate adventure and a roller coaster? Don't believe it, just try it out, if you have the chance.

Well, after having read all this, the factor three budget overruns and dysfunctional new systems that from time to time make it to the papers shouldn't surprise you anymore. The nice thing is that this need not be. It is the experience of the author that by abiding to the above simple common sense guidelines with respect to requirements, design, coding, testing and simulation, and by taking true responsibility for and pride in your work, problems like that can (almost) completely be avoided. Proactive behavior (Google) of you as a developer and intrinsic safety (Google) of the systems you help building, are what it takes.

# Constructor overloading and virtual destructors

Any method in C++ can be overloaded by specifying another method with the same name but with different parameter types. As a constructor is just a special method, this also holds for constructors.

By default each class has a parameterless constructor.

*class Matrix {*

*public:*

*...*

*private:*

*float \*\*contents;*

*}*

*// No constructor declared or defined anywhere, still this class has constructor Matrix ()*

The task of this parameterless constructor is to provide the object with a pointer to the virtual function table (Google). If you provide a constructor with a different prototype, e.g.

*Matix::Matrix (int nrOfRows,int nrOfColumns, float \*\*contents) {*

*...*

*}*

the parameterless default constructor will not be available anymore, since would it possible to instantiate a Matrix that doesn't know its size and contents.

You may then reintroduce the parameterless constructor e.g. as follows:

*Matrix::Matrix (): Matrix (0, 0, 0) {*

*...*

*}*

Note that in this case the parameterless constructor calls the constructor with parameters, in order to create a 0 x 0 matrix with no contents.

Yet another special type of constructor is the so called copy constructor, e.g.

*Matrix::Matrix (Matrix&matrix) {*

*/\**

*- Allocate dynamic memory for the contents*

*- Make the contents field of the object under construction point to that memory*

*- Copy the contents from the parameter into this dynamic memory*

*\*/*

*}*

The copy constructor has as its task: to copy the dynamic memory that field *float \*\*contents* of the parameter points to, into *this.contents*, the contents of the object under construction. Failure to define a copy constructor will result in Matrix objects being copied in an inadequate way. If there's no copy constructor, the following statement:

*Matrix matrix2 = matrix1;*

will just result the field *float \*\*contents* of matrix1 and matrix2 pointing to the same piece of dynamic memory (since the pointer is copied) rather than two separate matrices, each with their own piece of dynamic memory to hold their contents.

A copy constructor is needed in many not so obvious cases! Note the following function prototype:

*Matrix Matrix:: multiply (Matrix m1, Matrix m2) ...*

Calling this (very inefficient) function will result in three calls to the copy constructor, two calls to put the parameters on stack and one to copy the return value from stack to the caller. The following prototype for a copy constructor will not work :

*Matrix::Matrix (Matrix matrix) ...*

To pass the Matrix parameter, not by reference but by value, the copy constructor will be called, resulting in infinite recursion and a stack overflow. So copy constructors MUST have reference parameters. This is but a small example of how language facilities such as reference parameters, constructors and use of dynamic memory interlock to form a consistent language. Failure to understand references will lead to the inability to define a copy constructor, which will render the use of dynamic memory error prone at best. Also without understanding pointers or references, polymorphism won't work and arrays will be incomprehensible beasts, especially when passed as parameters. This is why this text is called "C++ essentials". The minimal language subset presented here is largely an "all or nothing" proposal. If you skip any of these essential concepts, you'll miss all the aspects that make this language such a powerful tool.

Since as you've seen in the example above, it is common to allocate dynamic memory in the constructor, the most logical place to delete it is in many but not all cases the destructor. But objects containing dynamic memory can be accessed via pointers or referenes to base classes, by virtue of polymorphism. In that case it is important that the right destructor be called, so not the one of the base class, that may not hold dynamic memory, but the one of the object that the pointer or reference actually points to. In other words: In that case the destructor must be a virtual function, e.g.

*virtual Matrix::~Matrix ();*

If the usage context of a class is unknown, it is advisable to make all public constructors and functions virtual, since an object of that class may come to be part of a polymorphic datastructure.

# Templates

C++ templates (Google) are a powerful generic typing mechanism, allowing you to write code for yet unknown types. Replacing such an unknown type by a "perfectly ordinary" (Stroustrup, creator of C++) known type happens before the "normal" compilation.

It would certainly offend some C++ adepts, but C++ templates are not a long way from simple macro's, literally replacing a placeholder by a known type. The difference is that replacing the unknown type by the known one happens automatically in many cases.

If a function has a template parameter, a function with the right parameter type is generated for each call signature:

*<template class Number>*

*Number add (Number n1, Number n2) {*

*return n1 + n2;*

*}*

is expanded to

*float add (float n1, float n2) {*

*return n1 + n2;*

*}*

if the call

*float x = add (3.14, 2.72);*

is encountered, and to

*int add (int n1, int n2) {*

*return n1 + n2;*

*}*

if the call

*int x = add (10, 20);*

is encountered.

Despite the suggestive placeholder typename *Number*,

*#include <string>*

*using namespace std;*

*string s = add ("fiets" + "pomp")*

will work as well, the add function being expanded to:

*string add (string n1, string n2) {*

*return n1 + n2;*

*}*

where the *+* stands for string concatenation.

Class templates should be "instantiated" explicitly, and as such are even more like macro expansion, at least superficially:

*template <class Number>*

*class Complex {*

*public:*

*Number realPart;*

*Number imaginaryPart;*

*}*

can be used to e.g. create complex numbers consisting of e.g. two floats or two doubles:

*Complex <float> complex1;*

*Complex <double> complex2;*

Since templates have to be expanded for all utilized types, sometimes the simplest way is to compile them together with the code that uses them, by putting them in the module header. Since the header is possibly included in many places, this leads to bloated executables, the same class being compiled multiple times.

One can also use a typedef to create a "perfectly ordinary" type:

*typedef Complex <float > FloatComplex;*

*FloatComplex complex3;*

Templates are only loosely related to object orientation. Many complex applications have been devised, including recursive templates and something called Template Metaprogramming. No need to look this up, no real need to use it as well. Use templates preferably in a simple way, to save duplicating sourcecode for different types.

One less desirable property of the use of templates is that error reports of the compiler tend to become very cryptic. Using the Standard Template Library (STL), in itself an asset, demonstrates this disadvantage in a rather dramatic way. C++ error reports are a weak point, mainly caused by the fact that C++ does not have an LL1 grammar (Google). C++ template code error reports are often completely incomprehensible at first sight.

The syntax of templates can look confusing and repetitive at first:

*template <class T>*

*Matrix <T>:: Matrix<T> (Matrix<T> &matrix) {*

*...*

*}*

Look up some examples on the Internet to get the hang of it.

# Operator overloading

Existing operators can acquire new meaning in C++, but priority and grouping rules stack intact, and an operator can only be overloaded for classes, not for built-in types.

An overloaded operator is just a function with a special name, allowing a special all format. Instead of

*Matrix multiply (Matrix m1, Matrix m2)...*

operator overloading can be used.

Member function

*Matrix Matrix::operator\* (Matrix m)...*

can be invoked as

*m3 = m1.operator\* (m2);*

but also as

*m3 = m1 \* m2;*

Free (non member) function

*Matrix operator\* (Matrix m1, Matrix m2)...*

can be invoked as

*m3 = operator\*(m1, m2);*

or as

*m3 = m1 \* m2;*

The free function is preferable, since an overloaded constructor of class Matrix could turn a number into a Matrix containing that number for all diagonal elements. Adding such a constructor would enable writing:

*m2 = m1 \* 3.;*

or

*m2 = 3. \* m1;*

With a member function the second assignment with 3 to the left of the \* would not work, since 3 is not a *Matrix*, hence the compiler would not know in which class to look for the appropriate constructor or \* operator. With a global function the function overload mechanism would attempt converting 3 into a *Matrix* by calling the appropriate *Matrix* constructor, since the type of the function parameters is *Matrix*. This would only work for matrices of fixed size, however. An alternative is to overload the *operator\**free function to accept floats as left or as right operand.

In the above overloads, everything is passed to *operator\** by value. This results in needless copying of possibly large matrices. Pass by reference would do a better job. The returned object poses an even larger problem. It cannot be returned by reference since it is on the stack and will be deleted if the function returns. Returning it by value, however, would result in needless copying by the *=* operator. The solution would be some kind of lazy evaluation: Matrices are only duplicated if they become different.

Even if no lazy evaluation is utilized, Matrix::*operator=* will have to be overloaded, since it should allocate dynamic memory and copy the contents of the matrix rather than the pointer. It's prototype should be *Matrix& Matrix::operator= (Matrix &matrix)*, to enable writing *(m1 = m2) = m3*. Note that this is not a typo, the C++ syntax allows such an expression so it should work, however pointless it seems. To that end the last statement of *operator=* is always *return \*this;*.

All in all properly overloading operators is far from easy and can lead to performance drops due to making unnecessary copies. It is often easier to just use functions, e.g. passing in memory for the multiplication as a reference parameter to avoid lifetime and duplication problems:

*void multiply (Matrix &result, Matrix &operand1, Matrix &operand2);*

# A glance at some low level facilities

C++ is suitable for large object oriented designs, but also for writing device drivers. This means that the language has a number of low level facilities. It is e.g. possible to keep a variable explicitly in register. While this so called *register declaration*, stems from times when processors were simple and caching was practically absent, the so called bit operations are indispensible on any processor when writing device drivers.

There are four binary (i.e. with two operands both interpreted as bit patterns) bit operators:

(The numbers in the examples are unsigned)

Bitwise and (masking): *&* e.g. *3 & 5 == 1*

Bitwise or: *|* e.g. *1 | 2 == 3*

Bitwise exclusive or: *^* e.g. *1 ^ 3 == 2*

Bitwise inversion: *~* e.g. ~254 == 1 (eight bit case)

And there are the bit shift operators:

Right shift: *>>* e.g. *16 >> 2 == 4*

Left shift: *<<* e.g. *16 << 2 == 64*

Furthermore it is possible to overload the *new* and *delete* operators to e.g. force placement in the address range that pertains to RAM rather than ROM memory.

By the use of *struct*, memory alignment can be controlled and a rich set of processor specific datatypes is usually at hand, making low level code efficient and compact. Compilers that are mainly used to generate code for PC applications tend to generate executables of kB's or even MB's for even the smallest program. Compilers for simple, microcontroller-like target platforms are far more economic in dealing with memory.

As IO facilities are not in the language itself, C++ code does not need an operating system to function. A machine language program directly compiled from C++ may very well be the only code required to drive a microcontroller.

On the other hand very large and still well structured C++ applications can be written, often portable over multiple operating systems. If a lot of high-level coding is involved, C++ and Python (or Cython) play very well together. This combination has become very popular in technical and scientific applications.

--//--