Programming Style and Optimisations - An Overview

Summary

In this lesson we introduce some of the style and optimization features you may find useful to understand as a C++ Programmer. Note however this is a voluminous subject occupying books of material. Use this information as a guide but take time to check out some of the suggested "Further Reading".

Although not essential to learning C++, it's important to understand the merits of producing readable code. You're also in the business of programming high performance software as a Games Programmer, so you need to know how to optimise code too. Following style guidelines and producing readable code will pay dividends during development, especially when it comes to debugging, integrating or extending your code. You also need to work as a team, so it is essential that your other team members can understand the code you have written.

Optimisations can give you that extra edge and squeeze the best performance out of the machine; but optimise wisely! That means; don't spend time at the beginning of a project optimising a function that is called once or twice. Optimise near the end of a project when the architecture of the software is set. Then target the functions which are called the most frequently—usually those inside loops. Not only will this approach deliver the most gain but you will avoid wasted effort and have a pre-optimised benchmark project to measure your changes against.

Function Types

It's important to understand the distinction between different function types in C++.

Functions A function is simply considered as an address by the compiler. When you call that function the address is obtained and execution proceeds from the memory address which that function has been allocated. The underlying assembly code will perform some book-keeping on the stack, but functions are "fast" with respect to execution time given their relative simplicity.

Methods C++ introduces the use of Classes; with Classes we have methods and virtual methods. Whenever the compiler encounters a (non-virtual) class method call during compilation, it adds an extra paramenter—a pointer to the class object. You can use this pointer explicitly; it's called the `this` pointer. Regarding performance however, the impact is negliable and shouldn't concern you.

Inline Functions and Methods If a function or method only contains few lines of code we can provide a "hint" to the compiler by marking such functions `inline`. If the compiler chooses to make a function inline, it replaces the function call with the function statements at compile time. This creates a larger executable as code is duplicated but inlining should make execution even faster.

Virtual Methods Inheritance and Polymorphism in C++ requires use of the `virtual` keyword. When program execution encounters a `virtual` method there is a slight performance penalty. This is due to the fact that classes possesing virtual methods require a `vtable` containing pointers to those methods. Basically, virtual methods are how `polymorphism` can operate at run-time but requires extra space for the `vtable`. The effort of indexing the `vtable` also has the potential to cause a costly data-cache miss.

By all means, use `virtual` methods; don't try to avoid inheritance and polymorphim. They were invented for a reason; programmers need them. That said, don't over-use virtual methods. There is a temptation sometimes to make every method in a base class virtual, in case a future class inherits

from it. Avoid this kind of thinking and try to restrict virtual methods to those methods of a class which are called least often.

Finally, note that the issues associated with virtual methods only apply when you call the method on a reference or pointer type of object. When you call a virtual method directly on an object (on the stack), the call is as quick as any non-virtual method call.

Function Parameters - Pass by Value Versus Reference

Along with Memory Leaks, you'll find this topic in just about any good book on C or C++. When you pass an object to a function by value, you incur a cost because the compiler must call the copy constructor and make a copy of the object. If the object is quite large and the function is called many times in your game loop, you may incur a significant performance penalty. Bear in mind—this is a problem with large objects; anything under 4 bytes (like an `int`) can be passed by value or returned with no extra overhead.

Whenever you pass an object as a parameter to a function, seek to pass that object by reference rather than by value; pass a reference or a pointer to the object. If the function or method should not modify the data in any way, then pass a `const` reference or pointer to make this fact explicit.

When returning data from a method or function, never return data that was created in the method by reference; it will be destroyed after the function returns. In these situations you need to return a copy of the data (by value), or use a pointer to the data and allocate it on the heap; data on the heap will persist after the function returns.

The Cache and Memory Alignment

Data Types If you've examined the primitive data types available, you will have noticed like `short int` type. Additionally, the `stdint.h` header file provides bit-designated types like `int16_t`, `uint16_t` and so on. You may be tempted to use these types as *standalone* variables. Note however that these may actually incur a greater overhead on machines where the word-size is greater than 16 bits. You will actually make savings by using a larger type, say `int` or `int32_t` on a 32 bit machine.

Cache Coherence Where it does make sense to use more compact types is when considering cache coherence. Nearly all architectures use caching to speed up operations of the processor. This means ideally, you want your program to bring in data from memory to the cache as infrequently as is possible. Using a `short` or `int16_t` instead of an `int` can provide better cache utilisation when they are "packed" into an array and stored contiguously in memory. In addition you can order the data fields of your classes so that the most regularly used data comes first in the class declaration. The theory is that these data will be brought into the cache when the object is accessed and reduce the number of cache misses.

This of course depends on algorithms which will use the full array of data when it is brought into the cache. Avoid multiple iterations over an array of objects which access one data field at a time, while you have the data in the cache, try and make use of it. More importantly, any benefit depends on memory alignment (see below).

Memory Alignment A cache stores data in cache lines—typically 32 bytes on most PC architectures. This means that 32 bytes at a time are brought into the cache. The memory address of any data brought into the cache must be a multiple of 32 on such systems. If your array of 16 byte integers are located in memory carelessly, more cache misses may be incurred than necessary.

Some compilers attempt to align data more carefully but it is worth familiarising yourself with any options your compiler provides when dealing with memory alignment. This refers to memory on the stack and memory you allocate dynamically on the heap.

In Summary Make some effort to understand some of your compiler's options and your machine's specifications, especially the Processor(s) Architecture and Cache Heirarchy—this isn't Java; you should understand the machine as a C++ Game Programmer.

Memory Allocation

Typically, games execute in scheduled games cycles relating to the frame rate; a time slice for AI, so many milliseconds for Graphics etc. This can be problematic when it comes to dynamic memory allocation because the time required for allocating memory on the heap is generally not constant but variable. It may be necesary to search for a free memory block. The heap is shared between threads so there may be some inter-process synchronisation. In addition, memory on the heap introduces the problem of memory-leaks.

Static Memory Allocation You can avoid the issues of dynamic memory allocation by avoiding the heap altogether. You can assign the memory you will use "statically" when the program starts and have it automatically reclaimed when the program terminates (thus avoiding the problem of memory leaks).

An example of this technique could have been employed in the "Matrix" class. Note that the class allocates the memory for the matrix on the heap so that the user can specify the dimensions. If we know beforehand that all matrices we create will be a pre-determined size, we could assign that memory statically and allow it to be reclaimed when the object is destroyed:

```
1 // constant elements
2    static const int default_x = 3;
3    static const int default_y = 3;
4 protected:
5    int cells[default_x][default_y];
```

static matrix

The main problem with this approach however is that memory is wasted when we claim more than required. In addition, our solution does not have the flexibility to cope with new memory demands.

Memory Pools A more satisfying solution to the issues of dynamic memory allocation is to employ your own "memory pool" objects. A memory pool is typically a class that creates a pre-determined amount of memory on the heap when initialised. Its role is then to manage that memory, granting portions of free memory to classes that use your memory pool, much like the regular calls `new` and `delete`. Because you would typically write your own memory pool you can control how memory is allocated and to which objects.

There are many implementation examples of memory pools on the web, ranging in complexity. In addition, the book "C++ For Game Programmers" provides a more basic example. Finally the "Boost" library for C++ provides a ready made implementation. Be aware there are issues when relying on 3rd party libraries in your game projects however—notably when it comes to compiling on architectures which do not possess those libraries.

In Summary It's important to bear in mind the issues related to Memory Allocation when writing your projects. However this does not necessarily leave you with a choice of either using Static Allocation or implementing your own Memory Pool. In some situations, simply moderating your use of the heap can be enough to avoid problems without incurring unnecessary complexity. For example, if a method returns an object, and that method is rarely called, you may consider returning a copy on the stack rather than creating a pointer to the heap (this will also avoid the possibility of a memory-leak).

Project Design

Encapsulation of Classes The class mechanism provides different levels of access specifiers, namely: public, private and protected. Designating some aspects of a class as private or protected, is designed to hide complexity from the user whilst guaranteeing a security mechanism for the class. When writing classes you're encouraged to take advantage of this feature and only expose the necessary methods of your class via the public specifier. In this way, if you make changes to the private functionality of the class, this is less likely to affect other classes in the project.

Generally speaking, you're advised to mark all your class' data fields "protected". If you want other classes to be able to access those fields, use "getter" and "setter" methods. This provides more control over which fields may be read and written to, which ultimately helps to identify and correct bugs in your code.

Design Patterns Design Patterns are primarily a style feature of Object Orientated Programming and not specific to C++. In essence, they describe approaches to class design, solving problems that feature recurrently in software projects. Although many "patterns" exist, you will probably find use for the following:

- Singleton—Allows you to stipulate that only one instance of a singleton object may exist; useful for file managers etc.
- Facade or Proxy Pattern—Provide a proxy for a class so that its implementation details can be altered without affecting the users of that class while it is "under construction".
- Observer Pattern—So you can register a set of observers for a particular "observable" class. If the observable class is changed in some manner, the observers are notified via call-backs.
- PIMPL Pattern—Stands for Private Implementation. You may also find this pattern by the name "Opaque Pointer" or "Cheshire Cat" pattern. This pattern, like the facade pattern, helps to hide implementation details from the user. It can have the additional effect of speeding up compilation times when implemented.

Examples of all these patterns exist on the Web. It is worthwhile acquainting yourself with them; once again "C++ For Game Programmers" provides descriptions and implementations of all the above patterns.

Project Organisation How you organise the files and classes in your project can have a big impact on maintaining your code, fixing bugs and dealing with excessive compilation times. Regarding compilation, large projects can easily feature compilation times in excess of 30 minutes. As you seek to correct bugs in your code, you will typically make a single "corrective" change and recompile your project to see the result. Clearly, if it takes over 30 minutes to compile your project this will become a nightmarish experience. Here are a selection of tips to help you manage your code and keep compilation times down:

- You should generally create a single header (.hpp) and source (.cpp) file per class.
- In your header files, aim to keep the number of "included" files to a minimum. Note that when a class A only requires a pointer or reference to class B, you do not have to include class B in class A's header file. In these situations simply provide a forward declaration of class B in the header file for class A; the syntax is `class B;`. Then move the include statement from the header file to the .cpp file of class A.
- Use Pre-compiled headers if possible. Different compilers offer different usage and options for precompiled headers, but these can significantly reduce compilation times in large projects so check out the rules for your own compiler.
- Organise your project into logical components such as Graphics, AI, Game Entities etc. The aim is to produce a modular project, where changes in one component do not require changes to others. This approach boosts development and testing when working as a team. Design Patterns can help here. For instance, where a class is required to use a class in a separate logical component, use the "facade" pattern so that the interface to the class remains unchanged while the implementation is completed. This will provide greater flexibility.

Project Development

There is a time for getting code to work and a time for optimizing. When you begin your project, it is generally considered good advice to focus on style and functionality rather than performance and optimizations. Begin by dividing the problem your project is addressing into smaller, more manageable sub-problems. Then focus on solving those sub-problems. Make the problem easier to manage by using simple data types or the simplest expression of the problem. For instance, if you need to sort a container of objects, begin with a container of integers. Get that working then adapt it for more complicated objects. Solve for one item of data, then two items, before you attempt to solve for an array of n items.

As you write your classes and solve your sub-problems, aim to keep your project flexible and comment your code—especially sections of unusual code—so you and your team know what's going on. Pay careful consideration of the algorithms you use and don't restrict yourself to one approach. For example, perhaps A* isn't the best choice for type of search algorithm in a particular scenario. Or maybe the data you wish to sort is almost sorted, making *Insertion Sort* more efficient than *Quick Sort*. Sometimes, if the algorithm is wrong, no amount of optimisations will help.

Finally, as mentioned in the Summary, don't labour over optimisations early on. You may find that you need to go back to something you solved earlier and make major changes. You may have to remove that function that you spent ages optimising. If so then you have wasted your time and effort. These early and mid-stages of development are where you should prioritise style and design over speed.

Further Reading

C++ is a complicated language—perhaps the most complicated you will ever use. We've attempted to cover the essentials but inevitably there are many more aspects of the language that you will need to grapple with in time. The following resources are recommended to help you:

1. C++ For Game Programmers by Michael J Dickheiser. Great book with plenty of material specifically for Games Programming. Assumes the author has basic knowledge of the language though.

2. Professional C++ by Solter and Kleper. A comprehensive and well-written book covering in-depth all the major language features, including Design Patterns, Distributed C++ and Integrating C++ with other languages.

The following web resource supplied by "Event Helix" provides information on a wide range of optimisations for C and C++ developers:

1. Optimizing C and C++ Code