High Performance Game Programming in C++

Paul Pedriana
Senior Software Engineer, Maxis
www.ccnet.com/~paulp/HPGP/HPGP.html

Introduction

At first, game programming was mostly done in assembly language, and occasionally even machine language. Then C came along and programmers could write most of the program in C, reserving assembly language for key bottleneck sections of code. Now C++ is becoming a prominent language, largely due to the advantages of the object-oriented paradigm. C++ didn't invent object-oriented programming; nor was object-oriented programming impossible with C. C++ simply provides a standard mechanism and support for it. With the ratification of the C++ language standard in 1997, that mechanism is now written in stone.

But games programmers are sometimes resistant to programming in C++. C++ does things behind your back, whereas C is straightforward. The C++ language standard is a moving target, whereas C has been fixed for years. C++ language syntax can get cryptic and complicated, leading to obfuscated and hard-to-debug code.

While C++ has suffered a reputation for being an unstable language, this reputation is largely unfounded. About 99% of the language has been set in stone for years. With the exception of `stringstreams`, every line of code we discuss here was completely legal and compilable code by most compilers by 1994 or 1995. We can feel pretty confident in the stability of the language from now on, since the language was ISO–standardized in 1997. I'm not saying that things are perfect, however. Our advice is to stay away from the most recent additions to the language, most of which are esoteric little things anyway. Some game development platforms, namely consoles, are a little behind PCs in support for C++. There has been good support of C++ on the PlayStation, but good Nintendo 64 support has lagged somewhat. In reality, you can get the GNU C++ compiler to work on most any platform.[1]

One thing that isn't mentioned by most of the C++ books out there is that many of the language's "implementation-dependent" mechanisms are in fact done the same way by most, if not all, compilers. For example, the definition of the C++ language does not specify how the virtual function mechanism is implemented, but instead leaves it up to the individual vendors. Well, the fact is that every compiler I've seen does it the same way (see the section on C++ structures and functions for the actual implementation). If you want to know more about how things are done internally, check out *More Effective C++…*, by Scott Meyers.[2]

Here are the topics we'll cover:

Writing and Benchmarking High Performance Code	Exception Handling
Inlines	Casting and RTTI
C++ Classes vs. C Structs	Stream IO
C++ Class Function Calls	STL
Memory Allocation via new and delete	C++ Strings
References	Templates
Operator Overloading	Conclusions

For conclusions we make in each topic, there is example code to demonstrate it available on the distribution CD and at the HPGP URL (see the top of this article). We're concentrating on the Intel processor and compilers here. There simply was not enough time to run all tests on other processors, such as the Motorola, PlayStation, and Nintendo processors. So if you want to draw definite conclusions for these processors, you'll have to take the example code and run the tests yourself.

Writing and Benchmarking High Performance Code

How good are compilers at optimizing code? This has been a topic of some debate for a number of years now. There are generally two views:

> Modern compilers are very good at optimizing code and can even out-optimize hand-coded assembly. Your best bet is to spend time improving your algorithms and leave the opcode optimization to the compiler.
>
> To get the highest-performing code, you should hand-code the bottleneck loops in assembly.

These two views aren't diametrically opposed, but the first one definitely puts some trust in the compiler while the second doesn't. There is no simple answer. I've found that the compiler sometimes does an excellent job of optimizing the code and other times it does a pretty poor job. Take the simple function below:

```
void DoTest7(int x, int* z){
   for(int i=0; i<x; i++)
      (*z)++;
}
```

This function is simple enough that you'd think the compiler would be able to optimize it properly. But it doesn't. Basically, it treats the variable `z` as if it was volatile—inside the loop it reads in `z`, increments it, then writes it out again. Not very efficient.

```
void DoTest7(int x, int* z){ // MSVC++ 5, w/ speed optimizations
   for(int i=0; i<x; i++)
                       mov    ecx,dword ptr [x]
                       test   ecx,ecx
                       jle    DoTest7+15h
       (*z)++;
                       mov    eax,dword ptr [z]
                       mov    edx,dword ptr [eax]
                       inc    edx
                       dec    ecx
                       mov    dword ptr [eax],edx
                       jne    DoTest7+0Dh
                       ret
}
```

In about five minutes, I rewrote the function to be like this:

```
__declspec(naked) void DoTest8(int x, int* z){
   __asm{
      mov    ecx,dword ptr [esp+4]
      test   ecx,ecx
      jle    Exit
      mov    eax,dword ptr [esp+8]
      mov    edx,dword ptr [eax]
   LoopTop:
      inc    edx
      dec    ecx
      jne    LoopTop
      mov    dword ptr [eax],edx
   Exit:
      ret
   }
}
```

The second version is 40% faster than the compiler-generated version. Obviously, the compiler didn't do a very good job this time. On the other hand, you might very well be able to argue that the example above is not very good–that the compiler could have simply said "`z+=x`" and returned. Well, OK. But nevertheless, I hold to my original statement: the compiler could have done better. Our recommendation is this: *always* check the disassembly of key bottleneck code produced by the compiler, and *always* time your code. You will hear this repeated a number of times.

What's the best way to measure your code's performance? Use the right tool for the job. Here are four measurement methods, from high to low level:

Frame Rate Inspection
Profiling
Precision Timing (microsecond or better)
Disassembly Examination

We're going to be spending the bulk of our time using the last two methods, since they are best for giving a microscopic view of your code. In practice, you'll need to employ all these methods to measure your code's performance.

Precision Timing and The Zen C++ Timer

Many of you will remember the classic book, *Zen of Assembly*, by Michael Abrash[3]. If so, then you'll also remember the Zen Timer. I now present the Zen C++ Timer: a C++ class called "MTimer" that performs precision microsecond-accurate timing. Here is the public interface:

```
class MTimer{
public:
   enum TimerResolution{nTimerResolutionMicroseconds,
                        nTimerResolutionMilliseconds,
                        nTimerResolutionSeconds};

   MTimer(TimerResolution nNewRes = nTimerResolutionMicroseconds);
   void                 SetResolution(TimerResolution nNewRes);
   TimerResolution      GetResolution();
   void                 Start();          //Starts timer.
   void                 Stop();           //Stops it. Doesn't reset it.
   void                 Reset();          //Stops timer and resets it.
   void                 Restart();        //Same as Reset(), Start()
   unsigned long        GetElapsedTime();
   bool                 IsTimerRunning();
};
```

The source code to a Windows version of this timer is available on the CGDC distribution CD and at the HPGP (High Performance Game Programming) URL listed at the top of this article. While the distribution examples for this article provide plenty of example usage, I'll give a simple example here.

```
Mtimer timer;
timer.Start();
DoSomething();
timer.Stop();
printf("DoSomething() took %u microseconds.\n", timer.GetElapsedTime());
```

On Intel CPUs, this timer uses the RDTSC (ReaD from Time Stamp Counter) instruction, available in Pentium and later processors. The RDTSC instruction is accurate to roughly 0.02 µs on 200MHz processors. We'll come back to the RDTSC instruction later in the **C++ Class Function Calls** section. Remember that while the Zen C++ timer is the best thing for precision timing of small sections of code, it isn't very good for high-level profiling. You're better off using a profiler like HiProf or Intel's VTune for this. Use the right tool for the job.

Under a multitasking operating system, you'll need to try to get all the processor time you can. Here are two ways to get maximum processor time.

```
__asm cli  //disable interrupts (and thus task switches, etc.)
<time your code here>
__asm sti  //enable interrupts
```

```
SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_TIME_CRITICAL);
<time your code here>
SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_NORMAL);
```

The former works under DOS and Windows95, but not under NT. Not without the appropriate privilege level, at least. Also, the code you are timing may actually need interrupts to be enabled in order for it to work properly. In that case, the latter solution will usually do fine. Empirical observation has shown that as long as your system is at idle and you don't do anything, including moving the cursor, you will get virtually all of the CPU time for brief periods with the latter method. For those who want to truly disable interrupts under NT, you need to write a kernel-mode driver that simply calls `cli`/`sti`.

You will also find that while setting up timing comparisons, it may be useful to set up a dummy function whose purpose is to accept a variable as anargument so that the optimizer doesn't optimize away that variable. Here's a good example of such a function, because it takes a void pointer and executes in a short but predictable period of time:

```
void DoNothing(void* ptrToSomething){    //Bogus function that does nothing.
   ::sprintf(" ", " ", ptrToSomething);
}
```

Inlines

Function inlining is one of the most important speed-enhancing features introduced by C++. While the C language definition doesn't have inlining, it is still implemented by some vendors as a language extension (e.g. `__inline` in MSVC) and it is likely to be a ratified addition to the next version of the C standard. Inlines are largely (though not entirely) intended to replace macros used in the C language. Inlined functions are cleaner, safer, and generally more flexible than macros. You commonly see inline functions in class Get/Set members and in template libraries such as the C++ STL. There are a number of questions we want to answer here: How well do current compilers implement inlining? How fast are inlines compared to macros? Is inline "just a suggestion" to the compiler that's unreliable, or do compilers actually inline the code you want? How many levels of inlining can you expect out of a compiler?

Modern compilers for the PC and Macintosh do a very good job with inlining. You can virtually rely on these compilers to inline code marked as `inline`. They will even inline other code as well if you let them (e.g. the /Ob2 argument to MSVC++). So then, is `inline` "just a suggestion?" Technically, yes. It is only a suggestion. However, unlike the `register` keyword, which is also merely a suggestion, the compiler is usually able to comply with your `inline` request. How fast are inlines compared to macros? Logically, with a decent compiler, they should be just as fast. But I've done a number of tests to make sure. In my tests, inline functions were *exactly* as fast as the equivalent macros, and the generated disassembly was identical as well. How many levels of inlining can you expect? In other words, if one inline function calls another inline function, what happens? Will the compiler inline both calls? In short, the answer is yes, at least with the compilers I've tested. In fact, I gave up after testing 12 levels and finding that the compiler still inlined all the calls. With MSVC++, there are some pragmas that provide some control over inlining recursion. Look up `#pragma inline_depth` and `#pragma inline_recursion` in the documentation for more.

"But," you ask, "isn't `inline` incompatible with `virtual`?" Let's back up a bit and review where the rationale for this comes from. Virtual functions are functions that can be overloaded in subclasses of a parent class, like this:

```
struct A{                                 struct B : public A{
   virtual void DoSomething();               virtual void DoSomething();
};                                        };
```

If you have a pointer to an object of class `B`, and you call `DoSomething()`, the version of `DoSomething()` defined in class `B` will properly get called. This works even if the `DoSomething()` is called through a pointer to

A. The reason it works is that, in virtually all implementations of C++, virtual functions are implemented with a "virtual function table," which is an array of pointers to functions. This is incompatible with function inlining because you simply can't have a pointer to inline code. Thus, for most situations, virtual and inline are incompatible; the compiler will simply ignore the `inline` statement. However, there is one situation whereby they are compatible: where the object is accessed directly instead of through a pointer.

```
struct B : public A{
   inline virtual void DoSomething(){ ::puts("Hi mom!"); }
};
B b;
b.DoSomething();    //Case 1: This call can in fact be inlined.
b.A::DoSomething(); //Case 2: This call can be inlined too. This is uncommon but legal
syntax.
```

In the example above, the compiler knows for sure in both cases what version of `DoSomething()` needs to be called (i.e. `B::DoSomething()`), and thus can inline the call. This sounds fine, but will a real-world compiler actually do it? You bet. Tests confirmed this with MSVC++ and in fact we'll see this same effect in the **C++ Class Function Calls** section later. On a side not, MSVC++ won't inline functions declared "naked" (`__declspec(naked)`).

C++ Classes vs. C Structs

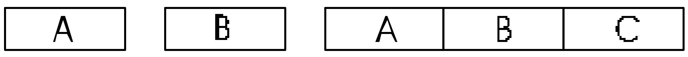
Classes are the very foundation of C++. Day one of every C++ class introduces classes. Page one of most C++ books introduces classes. What's the difference between a class and a struct in C++? There is only one: C++ class members are private by default and C++ struct members are public by default. As a result, there really isn't much reason to use `class` in C++ instead of `struct`. `private`/`protected` data is perhaps a little over-rated for some parts of game programming, but this is a very debatable topic.

In C++, classes (from now on I'll refer to classes and structs together as classes) are effectively a superset of C structs. C++ introduced the concept of POD (Plain Old Data) types. A POD type is a C++ class that acts exactly like a C struct. The question is, when does a C++ class *not* act like a C struct? The answer is basically this: when the class uses virtual functions and/or multiple inheritance. In the case of virtual functions, the class has an invisible data member that a C compiler would not expect or understand. In the case of multiple inheritance, the compiler shifts the position of the structs around. We'll take a look at this in a second. The take-home message is that C++ POD types are exactly interchangeable with C structs, are *exactly* as fast as C structs, and in fact can be passed to C-compiled code without a problem. For C++ non-PODs, data access is as fast as with PODS, but there will be an extra four bytes allocated for the virtual function table. We will discuss the table in the next section.

So what is this "position shifting" with multiple inheritance? To answer that, we need to look at how multiple inheritance is implemented. As with many other features of C++, the actual mechanism of multiple inheritance is implementation-dependent, but in fact is implemented the same way by most (all?) compilers. Consider the following classes:

```
class A{              class B{                class C : public A,public B{
   int a;                int b;                  int c;
};                    };                      };
```

This is what these classes look like in memory when you instantiate them:



What this is showing is that in memory, an object of `class A` is simply a four byte `int`, and a pointer to `A` is also simply a pointer to the `int a`. The same goes for an object of `class B`. However, `class C` is a composite of `A`, `B`, and the members of `C`. It is implemented by simply placing `A` and `B` end-to-end, and then tacking on the `C` members to the end of that. Thus, when you have a pointer to a `C`, it is actually also a pointer to the `A` within `C` (see the picture above). If you have a pointer to a `C` and `static_cast` it to a pointer to a `B`, the compiler adds four bytes to the pointer to `C`, because the copy of `B` within `C` is shifted by four bytes, or `sizeof(A)`. Thus, while `class C` is also a `class B`, they are not interchangeable—you must use a C++ `static_cast` to convert between them. See the section below on casts for more on this. This situation does not occur with single inheritance. With single inheritance, the subclass is simply a *continuation* of the parent class, and not a composite of anything.

C++ Class Function Calls

One of the cornerstones of our investigation of C++ features is the discussion of C++ class function calls. How expensive is a class member function call relative to a standard C function call? What about inline functions, virtual functions, and static functions? In the paragraphs that follow, I will use the term "global function" to refer to a function that doesn't belong to any class. Thus, all C-like functions are global functions.

C++ member functions can operate on member data because they add a hidden parameter to every function: the "this" pointer. For example, if you declare a C++ member function that takes a single `int` argument, the compiler actually generates a second argument which is the pointer to the object itself.

You say this:	The compiler generates this:

`class A{`
`    void DoA(int x);`
`};`
	`class A {`
	`   void DoA(A* a_ptr, int x);`
	`};`

This way, the function knows how to access the member data of A. This is very much like C library programming. Let's take BSD sockets as an example. Here are some of the functions in that library:

```
int recv(SOCKET s, char* buf, int len, int flags);
int send(SOCKET s, const char* buf, int len, int flags);
int setsockopt(SOCKET s, int lvl, int opt, const char* val, int len);
```

Notice that the first parameter is always a `SOCKET`. This is much like how C++ implements classes, as we saw above. C++ hasn't invented anything new, it has simply standardized a way of doing something people have been doing for years. Let's look at a disassembly of the actual passing of the "this" pointer.

You say this:	The (Microsoft) compiler generates this:

`A a;`
`a.DoA(3);`
	
	`push        3`
	`lea         ecx,[a]`
	`call        A::DoA(0040c300)`

As you can see, the pointer to a was in fact passed in the `ecx` register, and not on the stack. This is slightly faster than if it was actually passed using the stack. The compiler can make this assumption because both the caller and the function itself know that this is a C++ class and have agreed ahead of time that the "this" pointer always gets passed in the `ecx` register. However, this only happens with the Microsoft Compiler on the Intel platform. Virtually all compilers have an option to pass other parameters in registers as well. With PC compilers, it is the `__fastcall` option (available on most PC C++ compilers). I recommend using `__fastcall` for most situations, though in practice you'll hardly be able to notice the speed improvement except for small functions. The problem with `__fastcall` is that it makes plug-in or DLL programming difficult. If your DLL uses one compiler or calling option, and your main executable is using another compiler and `__fastcall`, you will have problems.

What about `inline` member functions? Since the code is inlined, there is no need to pass any "this" pointer, since the data is manipulated directly.

Virtual functions are one of the main draws of C++. A virtual function lets you override the behavior of a parent class in a subclass. Mostly likely, you know all about this already. But how does it work? What happens under the hood? Every class that has a virtual function declared for it has a hidden data member called the "vTable pointer" (virtual function table pointer). This is a pointer to an array of function pointers. This hidden data member is always implemented as the first data member of the class, in the case of single inheritance. See the previous discussion on multiple inheritance for how it's done there.

You say this:	The compiler generates this:
`class A{`	`class A{`

```
   int a;                              void* vTablePtr; //Points to DoA1, DoA2
   virtual void DoA1();                int   a;
   virtual void DoA2();             };
};
```

Note that on the right-hand side above, I don't include the function names. This is because in reality, the actual object data only consists of the two members shown. The function declarations on the left are only declarations; they don't take up any space on a per-object basis. If you were to get the `sizeof(A)` it would return a size of eight bytes. Only classes that have virtual functions will have this hidden vTable pointer. For every class (not for every *object*) that has a vTable pointer, the compiler creates a single vTable. The vTable pointer is set when the object is constructed. There is further discussion about this in the next section (**Memory Allocation via `new` and `delete`**). Here is an example of code generation with virtual functions:

You say this:	The compiler generates this:
`A* pA = new A;`	`<asm to create A not shown>`
`pA->DoA1();`	`mov   edx,dword ptr [edi] //Note that pA is`
	`mov   ecx,edi             //stored in edi.`
	`call  dword ptr [edx]     //1st table entry`
`pA->DoA2();`	`call  dword ptr [edx+4]   //2nd table entry`

pA is put in the `edi` register. Since the vTable pointer is the first member of the object, the compiler can load the vTable pointer into `edx` by simply saying: "`mov edx, dword ptr [edi]`." Since DoA1 and DoA2 are the first and second entries in the function table, you can see how their pointers are loaded above.

But how fast is this? To compare virtual function access with non-virtual function access, I set up a number of classes with virtual, non-virtual, and static functions. Also, a couple global C functions were tested as well. When using a standard microsecond timer to measure the differences between these, you get basically the same answer for all of them, leading you to conclude that a C++ virtual function call takes just as long to execute as a non-virtual function. But it's pretty hard to measure such small differences in microseconds, so we resort to the lowest-level timer there is: the CPU clock tick counter. This is the RDTSC instruction on Pentium and later x86 CPUs (not available on some clone CPUs). The PC RDTSC instruction increments once for every CPU clock cycle. Here's how you time code under MSVC++ with the RDTSC instruction:

```
#define rdtsc __asm _emit 0x0F __asm _emit 0x31
unsigned nStart, nStop, nResult;
rdtsc                                //Read clock ticks into edx:eax
__asm mov nStart, eax                //Save the result
<put the code you want to time here> //
rdtsc                                //Read clock ticks into edx:eax
__asm mov nStop, eax                 //Save the result
nResultTicks = nStop-nStart;         //Calculate the sum.
```

The MSVC++ 5 inline assembler doesn't understand the `rdtsc` instruction, so we must emulate it with a macro. The same exists for Borland C++. With BC++, you must do this:

```
#define rdtsc __asm DW 310F //Borland inline assemble version of rdtsc
```

Enough of that. Now here are the results, which were quite consistent on both of the testing machines used:

Function	Usage	Ticks: Pentium	Pentium II
`void DoIt();`	`DoIt();`	28	41
`void DoIt(struct*);`	`DoIt(ptr);`	28	41
`void A::DoIt();`	`a.DoIt();`	28	41
	`a->DoIt();`	28	41
`void A::DoIt(int);`	`a.DoIt(1);`	33	44
	`a->DoIt(1);`	34	44
`virtual void A::DoIt();`	`a.DoIt();`	28	41
	`a->DoIt();`	31	53
`virtual void A::DoIt(int);`	`a.DoIt(1);`	33	44

	`a->DoIt(1);`	35	56
`static void A::DoIt();`	`A::DoIt();`	29	41
	`a.DoIt();`	27	41

We can sum these results by saying that on a Pentium II there is a slight cost to a virtual function call through a pointer. On a Pentium, there is almost no cost, if any at all. Why is there a cost in going through "`->`" and not through "`.`"? See the end of the **Inlines** section above for the answer. In either case, the cost differences are likely to get lost in the noise, especially with regard to the state of the L1 cache. In reality, we can't draw any simple, concrete conclusions from one set of experiments like this, and we don't have enough space here to do a thorough analysis of all of this. Read the article, "Pentium Processor Optimization Tools"[4] for details on Pentium pipeline performance issues.

A C++ `static` member function has visibility into a class's data members, but it doesn't execute in the context of the class. `static` functions can't access class member data unless you give them a class to work on. There is no hidden 'this' pointer, as with non-static class member functions. As such, `static` functions are just like global functions. And while they, unlike global functions, have visibility into a class's private members, they otherwise are treated by the code generator just like global functions. Here are a global C function and a C++ class static function compared side-by-side:

Global function call	C++ static member function call
`int y;`	`int y;`
`void DoIt(){`	`class X{`
`   y++;`	`   static void DoIt(){ y++; }`
`}`	`} x;`
`DoIt();`	`X::DoIt();`
	`x.DoIt();`
`1: void DoIt(){`	`1: void X::DoIt(){`
`     inc dword ptr [0041c5d8]`	`     inc dword ptr [0041c5dc]`
`2:   y++;`	`2:   y++;`
`     ret`	`     ret`
`3: }`	`3: }`
`4: DoIt();`	`4: X::DoIt();`
`      call DoIt(0040c2c0)`	`     call X::DoIt(0040c320)`
	`5: x.DoIt();`
	`     call X::DoIt(0040c320)`

Notice that the code generated by the compiler in both cases is identical. Also note that accessing the static member function through `.` or `::` acts the same.

That concludes our discussion on C++ function calls. The take-home message is that member function calls are so inexpensive, especially with the default of passing the "`this`" pointer in a register, that the cost of using them is negligible for 99% of the code that you'll write. When you want the fastest possible code, such as for bottleneck sections, you generally will want to bypass function calls altogether and implement your code with direct accesses as much as possible. However, as we've already seen, C++ inline functions are *exactly* as fast as direct accesses, so you can still use function calls in bottleneck sections.

Memory Allocation via new and delete

How fast are `new()` and `delete()` compared to `malloc()` and `free()`? As we will see here, they are *exactly* as fast, and provide some extra benefits as well. But first, let's take a look at what happens under the hood when you create on object via `new()`. Consider the following class:

```
class A{
   A(){ printf("In  A()\n"); }
  ~A(){ printf("In ~A()\n"); }
};
```

When you create an object of `class A` via `new()`, here is how most compilers implement it:

You say this:	The compiler generates this:

`A* pA = new A;`	`pA = malloc(sizeof(A));`
	`pA->A();`

When you delete an object of `class A` via `delete()`, here is how most compilers implement it:

You say this:	The compiler generates this:

`delete pA;`	`pA->~A();`
	`pA = free(pA);`

But what if you don't want to have the overhead of calling the constructor? What if you want a "lean and mean" class that creates and destructs as fast as a simple C struct? That answer is that you simply don't declare a constructor and destructor. If you don't declare a constructor the compiler will call `malloc()` and be done with it. Many of the C++ texts, including the language standard itself, will tell you something like this: "If you don't declare a specific constructor and destructor for a class, the compiler will generate a default one for you." Well, the truth is that all known optimizing C++ compilers will simply generate nothing for virtually all cases. The compiler will not, for example, create some invisible function that sets all data in the object to zero. There is a slight exception in the case of classes with virtual functions, as we will see below. In other words, if you don't declare a constructor, calling `new()` will simply result in the memory allocation and nothing else—*exactly* the same as in C. Destructors work the same way as constructors, but do things in reverse order.

What happens in the case of inheritance? How do the constructor calls get chained? Consider the following classes:

```
class A{                            class B : public A{
   A(){printf("In A()\n");}            B():A(){printf("In B()\n");}
};                                  };
```

For most compilers, this is what happens:

You say this:	The compiler generates this:

`B* pB = new B;`	`pB = malloc(sizeof(B));`
	`pB->B();`

So how does the constructor for A get called? The answer is that the constructor for B calls the constructor for A, as shown in the box below. Multiple inheritance works basically the same way.

You say this:	The compiler generates this:
`B():A(){printf("In B()\n");}`	`B(){A(); printf("In B()\n");}`

This is pretty simple, isn't it? What happens in the case of virtual functions? How does the virtual function table get set up? Since every C++ object has a specific virtual function table associated with it, it makes sense that something must happen during `new()` that sets up that table. The answer is that the constructor for a class sets the virtual function table pointer. Consider the following classes:

```
class A{                            class B : public A{
   A(){printf("In A()\n");}            B():A(){printf("In B()\n");}
   virtual void DoSomething();         virtual void DoSomething();
   long a;                             long b;
};                                  };
```

Since a virtual function is declared for these classes, a virtual function table pointer must be made. Thus, the classes actually look like this in memory:

```
class A{                            class B{
   void* vTablePtr;                     void* vTablePtr;
   long  a;                             long  a;
};                                      long  b;
                                    };
```

Note that while B is a subclass of A, it is implemented in memory as if it *contains* a copy of A as its first data. This is how virtually every C++ compiler implements inheritance. Note that class B has only one virtual function table pointer, as opposed to two. Now we are ready to answer the question of how the virtual function table is set up in constructors. For the above classes:

You say this:	The compiler generates this:

`A(){printf("In A()\n");}`	`A(){`
	`   vTablePtr = <address of vTableA>;`
	`   printf("In A()\n");`
	`}`

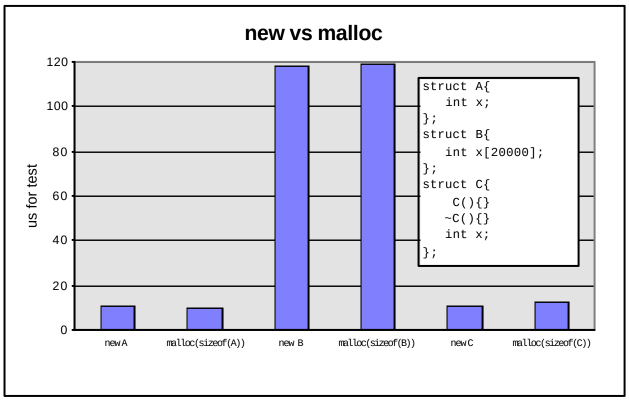
`B():A(){printf("In B()\n");}`	`B(){`
	`   A();`
	`   vTablePtr = <address of vTableB>;`
	`   printf("In B()\n");`
	`}`

Note that the constructor for `B` calls the constructor for `A` *before* it assigns the virtual function table pointer of `B`. That way, when `B()` is finished, the proper virtual function table pointer will be in place. Note that if you didn't declare a constructor for `class A` or `B` above, the compiler still has to generate code to assign the virtual function table pointer. What this amounts to is a simple assignment after calling `malloc()`.

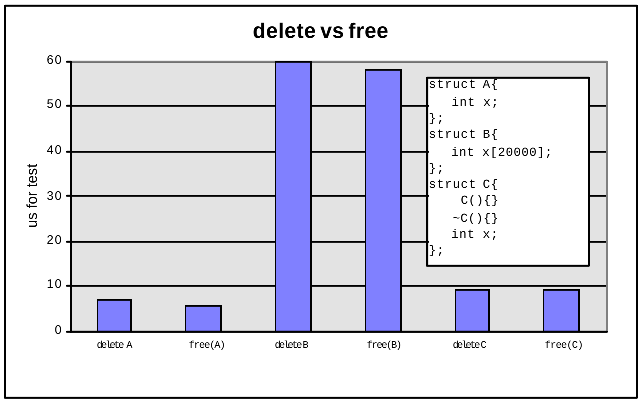
One last thing you may be wondering about is, "What about objects created as automatic variables on the stack or global space instead of in the heap via `new()`?" The answer is that they work *exactly* as with objects created via `new()`, except the call to `malloc()` is removed.

We now have a pretty good picture of just about everything that happens under the hood when you use `new()` and `delete()`. With this knowledge, you should be able to predict what the compiler will actually do when you create C++ objects. Now I present some benchmarks to demonstrate what we discussed above. The basic result is that C++ acts just like C in the absence of virtual functions and inheritance, and adds a very slight overhead otherwise.

Here are some benchmarks that show what I'm talking about:



```
struct A{
   int x;
};
struct B{
   int x[20000];
};
struct C{
    C(){}
   ~C(){}
   int x;
};
```



```
struct A{
   int x;
};
struct B{
   int x[20000];
};
struct C{
    C(){}
   ~C(){}
   int x;
};
```

There is one last thing I want to talk about: custom allocators. C++ lets you declare custom allocators on a class by class basis, and globally as well. This is great for game developers because we often need to implement things like dynamic lists that need to be lightning fast at allocation and deallocation of list items. With this mechanism, you can call `new()` for a class and it will use your custom fast allocator instead of the relatively slow runtime library version. You can certainly write a custom allocator in C for a given struct, but you must replace all calls to `malloc()` for that struct with the name of your custom allocator function. This is certainly a feasible solution, so I'm not here to criticize it. However, using the overloaded `new` and `delete` operators lets the compiler deal with the inheritance and virtual function table stuff transparently. Here is how you implement a custom allocator for a class:

```
class A{
   A() { printf("In A()\n"); }
   static void* operator new(size_t size);
   static void  operator delete(void* ptr);
```

```
    };
```

```
void* A::operator new(size_t size){
   //Allocate the memory here and return a pointer to it
}

void  A::operator delete(void* ptr){
   //Free the memory located at the pointer
}
```

References

References are an alternative to pointers as a method for accessing an object indirectly. A reference to an object is an alias to an object. Unlike a pointer, a reference cannot be changed after it is initialized. Nor can a reference be initialized to a constant (more exactly, a non-lvalue).

```
int x, y;
int& refX = x;
refX++;
refX      = y;  //This doesn't reassign the reference to refer to
                //  'y', it simply assigns the value of y to x.
int& ref1 = 1;  //Compiler error. Can only assign a reference to
                //  something that can be addressed (an lvalue).
int& refNone;   //Compiler error. Can only assign a reference to
                //  something.
```

Reference arguments can, for example, be used in place of pointers like this:

```
void DoSomething(int& x){          void DoSomething(int* x){
   x = 5;                             *x = 5;
}                                  }
```

The two functions above act exactly the same. In fact, as we will soon see, they generate the *exact* same disassembly. Many people wonder why references were added to the language, since they seem so much like pointers. The main reason is that references allow operator overloading to work smoothly. An example will help explain this. To overload the '+=' operator for a class, you use this syntax:

```
struct X{
   int n;
   X& operator+=(int i) { n+=i; return *this; }
}
```

This allows us to write code like the following:

```
X x1, x2;
x2 = (x1 += 2);
```

This code can only compile efficiently if we define `operator+=` to return X&. If instead we defined the `operator+=` to return X instead of X&, the compiler would have to (inefficiently) make a temporary copy of X between the two assignments. But we are getting ahead of ourselves here. Operator overloading is the topic of the next section.

The take-home message about references is that they are exactly as efficient as pointers, because from a code generation standpoint, they are identical. Here's the proof, generated by MSVC++ 5:

```
7:    void PtrTest(int x, int* z){
8:       for(int y=0; y<x; y++)
9:          (*z)++;
         mov         ecx,dword ptr [esp+4]
         test        ecx,ecx
         jle         PtrTest(0x00401234)+14h
         mov         eax,dword ptr [esp+8]
         mov         edx,dword ptr [eax]
```

```
          inc         edx
          dec         ecx
          mov         dword ptr [eax],edx
          jne         PtrTest(0x0040122c)+0Ch
          ret
10:   }
```

```
13:   void RefTest(int x, int& z){
14:      for(int y=0; y<x; y++)
15:         z++;
         mov         ecx,dword ptr [esp+4]
         test        ecx,ecx
         jle         RefTest(0x00401254)+14h
         mov         eax,dword ptr [esp+8]
         mov         edx,dword ptr [eax]
         inc         edx
         dec         ecx
         mov         dword ptr [eax],edx
         jne         RefTest(0x0040124c)+0Ch
         ret
16:   }
```

References have their advantages and disadvantages. They have the advantage of making code easier to read and allowing object operator chaining operations. They have the disadvantage of not being able to hold extra information, like a pointer can. By this, I am referring to the fact that a pointer can be set to `NULL` before being passed into a function as a message to the function to ignore the pointer or use some default behavior, and so on. You simply can't do this with references. Nevertheless, I find references to be nice and use them as often as pointers.

What really happens when you say "`return *this`"? The answer depends on whether you are returning a reference to the object or returning the object by value. If you are returning an object by reference, then a pointer to the object is simply returned. If you are returning by value, then the standard C/C++ return-by-value mechanism occurs, in which a hidden pointer is usually passed to the function (yes, C has hidden function parameters!). If the function is implemented inline, then the compiler effectively ignores the actual "`return *this`" statement.

Operator Overloading

Operator overloading allows you to redefine operators for an object. Operator functions for a class can both be inherited and overridden in child classes. A notable exception is `operator=()`, which can't be inherited by a subclass; each subclass must provide their own implementation. This makes some sense, because with the child, the assignment target is a different class—one that the parent can't know how to construct. Here is a list of all operators you can redefine:

```
+        -        ++       --       *        /        %
^        &        |        ~        !        <        >
=        +=       -=       *=       /=       %=       ^=
&=       |=       <<       >>       >>=      <<=      ==
!=       <=       >=       &&       ||       ->*      ->
,        ()       []       new      delete   new[]    delete[]
```

As we saw in the inlining section, inline functions manipulate data as fast as either direct access or macros. This applies to operator overloading as well. Consider the following class:

```
struct A{
   int a;
   A& operator ++() { ++a; return *this; }
};

A testA;
++testA;     //These two versions of ++ are just as fast as each other.
++testA.a;
```

Here's the disassembly:

```
30:      ++testA;
```

```
      mov         ecx,dword ptr [esp+0Ch]
      inc         ecx
      mov         dword ptr [esp+0Ch],ecx
32:      ++testA.a;
      mov         eax,dword ptr [esp+8]
      inc         eax
      mov         dword ptr [esp+8],eax
```

If the operator is defined as a function instead of an inline, then a function is called and the operation is predictably slower. If you need the overloaded operator to do more than just a simple increment operation, then you probably would make it a non-inline function. In that case, the extra code involved in the function may make the function call overhead negligible. It's your choice, you can do it either way.

Before I give you too rosy an impression of operator overloading, I must warn you about one potential issue: intermediates. Consider the following code:

```
struct B{
   B()      { printf("B::B()\n");         }
   B(int n) { b=n; printf("B::B(int)\n"); }
  ~B()      { printf("B::~B()\n");        }
   int b;
};
B operator +(const B& b1, const B& b2){
   return B(b1.b + b2.b);
}

B b1(1), b2(2), b3(0);  //Three objects created here.
b3 = b1+b2;             //One intermediate object created here!
```

Here's what actually prints:

```
B::B(int)
B::B(int)
B::B(int)
B::B(int)
B::~B()
B::~B()
B::~B()
B::~B()
```

For every "+" call, an intermediate object must be created. This may be quite inefficient. Note that an intermediate is created for the "+" operation above, but not for the "++" operation discussed previously. So how do you know when you're going to be generating intermediate objects? Easy: when the operator function creates one. The "++" operator above simply increments and returns "`*this`", whereas the '+' operator creates and returns an instance of "`B`".

We will see more on operator overloading efficiency in the section on STL.

Exception Handling

True exception handling has been considered a sorely needed addition to the C language for "mission-critical" development. History has shown that other languages, such as Ada, are better for this kind of task. C's and C++'s power, and hence their popularity among game programmers, is that they let you write code that's pretty close to the machine itself. The language definition also allows very easy interfacing to assembly code as well. Exception handling, however, is a mechanism that doesn't map very simply to any low-level machine operation. Perhaps that's why it wasn't originally designed into C.

How does exception-handling work? Technically, the exception-handling mechanism is "implementation-specific." But, like most of C++'s "implementation-specific" features, 90%+ of the compilers implement exception handling in more-or-less the same way. The differences often come in if and how the exception handling mechanism interacts with the operating system. We have only enough space here to briefly examine the mechanism. For a good article on the details of how exception handling is implemented by C++ compilers under Win32, see Matt Pietrek's article in MSJ.[5]

Basically, when you enter a function, a structure that contains the address of code that destroys stack-based objects is pushed onto the stack. The setup, push, pop, and shutdown of this structure takes up a number of machine instructions. However, with any decent compiler, this setup code only gets generated if the function creates automatic objects that have explicit destructors. Our experience has shown that most of the time-critical functions in a game app don't create these kinds of objects, and thus tend to not have this extra setup code.

Consider the following code:

```
struct A{
    A();
   ~A();
};
void TestFunction(){
   A a;
}
```

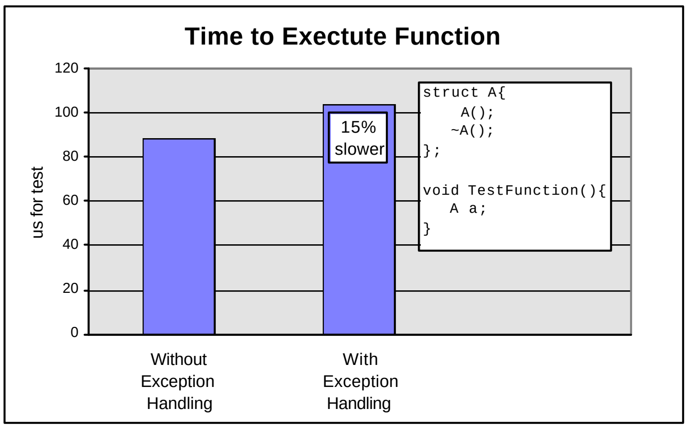
Here is the disassembly of the function with exception handling disabled:

```
7:     void TestFunction1(){
   push      ebp
   mov       ebp,esp
   sub       esp,00000004
8:        A a;
   lea       ecx,dword ptr [a] //Notice here that the compiler is putting
   call      A::A (00401290)   //the "this" pointer in the ecx register
9:     }                       //instead of pushing it onto the stack.
   lea       ecx,dword ptr [a]
   call      A::~A (004012a0)
   mov       esp,ebp
   pop       ebp
   ret
```

Here is the disassembly of the function with exception handling enabled. The highlighted code is the code added by the compiler to support exception handling:

```
7:     void TestFunction1(){
   push      ebp
   mov       ebp,esp
   push      ffffffff
   push      00401304
   mov       eax,dword ptr fs:[00000000] // Save the previous exception
   push      eax                         // registration structure.
   mov       dword ptr fs:[00000000],esp // Push the current exception
   sub       esp,00000004                // registration handler for
8:        A a;                           // this function.
   lea       ecx,dword ptr [a]
   call      A::A (004013b0)
9:     }
   mov       dword ptr [ebp-04],ffffffff
   call      $L25934 (004012fc)          // Jump down to the destructor
   mov       eax,dword ptr [ebp-0c]
   mov       esp,ebp
   mov       dword ptr fs:[00000000],eax // Restore previous exception
   pop       ebp                         // registration handler.
   ret
$L25934:
   lea       ecx,dword ptr [ebp-10]
   jmp       A::~A (004013c0)
$L25933:
   mov       eax,0040dc20
   jmp       ___CxxFrameHandler (004014d0)
```

Here are the results of timing the two versions:



For most game programmers, this is just too many instructions to spend preparing for something that almost never happens. Yet there are times when exception-handling is very useful, such as when you are doing a number of divisions and don't want to have to compare the denominator to zero before every one. You can simply let the exception handling mechanism catch the rare cases. So what is one to do? It turns out that some compilers allow a happy medium: exception handling enabled but automatic variable destruction stack unwinding disabled. With MSVC++, you simply disable exception handling's stack unwinding in the compiler options dialog box or on the command line with "`-GX-`" (the dash at the end means "disable"). This way, you can use `try`…`catch` statements, but when an exception happens, the `catch` works, but stack unwinding doesn't happen.

My recommendation for most situations is to use exception handling only sparingly. Use `try`…`catch` statements where you "expect" exceptions to happen, and not as a replacement for function return values or as a replacement for `gotos` (you read it right), and definitely not to catch programming bugs. I've found that when programmers get carried away with exception handling, the code becomes hard to maintain, despite the fact that it may be "academically correct." What we like to do at Maxis is put one `try` statement around the entire game loop. If an exception happens, then give the user an option to save the game, and quit the app. Experience has shown that 90% of the time, the game save works fine. Users love this.

One last important thing to know about C++ exception handling: it doesn't always work! You heard it right. Consider the following code:

```
void FunctionThatTrashesExceptionHandling(){
   A   a;     //'A' is a class that has an explicit destructor.
   int x[4];
   for(int i=0; i<40; i++) //Trash the stack.
      x[i] = 0;
   *((char*)0) = 0; //Do something that causes an exception.
}
try{
   FunctionThatTrashesExceptionHandling();
}
catch(...){
   printf("This catch will never happen!\n");
}
```

The program will bomb and the exception will never get caught (at least it didn’t on the Windows95 test machine). Because the exception handling mechanism uses the stack, the mechanism will fail if the stack gets corrupted. This lends credence to the belief that exception handling shouldn’t be used to catch bugs.

Casting and RTTI

Here are the five types of casting available in C++:

C-style cast (just like in C)
In C++, the C cast acts one of two ways. If the compiler knows that the source and dest types are related through inheritance, then the compiler makes the C cast implicitly act exactly like `static_cast`. If the two types are unrelated, then the compiler makes the C cast implicitly act exactly like `reinterpret_cast`. Unless multiple inheritance is involved, there is no speed hit. With multiple inheritance, there may be a small speed hit. See below for an explanation.

`reinterpret_cast`
This means, "Hey, this is a pointer to the type I'm saying it is. Trust me." Thus, `reinterpret_cast` on C++ classes acts just like C casts on structs in the C language. No speed hit.

`static_cast`
This cast can only be used to convert between types that you know are related in an inheritance chain. The compiler does type checking for you when you use this cast, which is nice. If C inherits from B, then you can use `static_cast` to convert a B* to a C* and vice versa. There's another reason to use this kind of cast instead of reinterpret_cast: when multiple inheritance is involved. In this case, `static_cast` is necessary, or the code will bomb. This is due to the way multiple inheritance works. See below for more on multiple inheritance. Unless multiple inheritance is involved, there is no speed hit. With multiple inheritance, there may be a small speed hit. See below for an explanation.

`dynamic_cast`
This cast allows you to take a pointer to an object and treat it as an object of the given type, based on the actual run-time type of the pointer. `NULL` is returned if the cast cannot be made. See below for a description of the mechanism. Significant speed hit, and some memory hit as well.

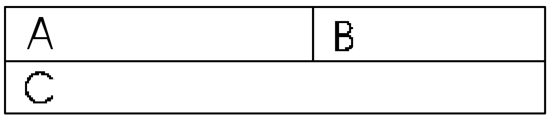
`const_cast`
This cast simply makes an object that is constant be non-constant. No speed hit.

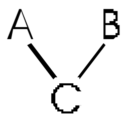
If we examine the disassembly of code that uses each of these types of casts, what we see is that except for two cases, all the C++ casts map to the same mechanism (i.e., no mechanism) as the C-style cast. See the HPGP URL for the actual disassembly. Thus, for most C++ casts, no code is generated at all -- the pointer is simply used as is. Here are the other two cases:

C cast/`static_cast` with multiple inheritance
`dynamic_cast`

C cast/`static_cast` with multiple inheritance

C++ implements multiple inheritance by simply storing the parent class objects sequentially in the subclass. Thus, if you have class `C` that inherits from both `A` and `B`, class `C` is implemented by simply putting `A` and `B` after each other in memory and returning the pointer to `A` and calling it `C`. Thus, if you have a pointer to a `C` and need to cast it to a pointer to a `B`, the pointer returned by `static_cast` will be the pointer to `C+sizeof(A)`. However, C++ also stipulates that if the pointer to C is `NULL`, then the pointer to B that results from the `static_cast` must also be `NULL`. Thus, when you are casting `B` to `C` or `C` to `B`, the generated code will compare to `NULL` first. Note that casting `A` to `C` or `C` to `A` will not generate the comparison to `NULL` nor the pointer movement described above. This is because `A` and `C` start at the same place within `C`.





Class C is simply the composite of A and B laid back-to-back.

dynamic_cast

Dynamic casting, part of C++ Run Time Type Identification (RTTI), uses the object's virtual function table to identify the object's exact class. Since for every class that has a virtual function table there is exactly one table per class, we can store class-specific data in this table as well. One such class-specific data item is the class identification itself. `dynamic_cast` simply pushes some pertinent parameters on the stack and calls some compiler-specific routine which reads some information out of the class-specific data. One side effect of this system is that only classes that have a virtual function table can implement RTTI. Also, since code must be generated on a class-by-class basis, the program size will increase as well.

Stream IO

When you first start learning C++, your teacher or the book you're reading often starts off by showing you these two things called `cout` and `cin`. Many of the console application examples will look something like this:

```
void main(){
   cout << "hello world!" << endl;
}
```

I don't know about you, but I never liked this stuff. Yet for some reason, most C++ authors think C++ stream IO is the greatest thing and that printf() and its brethren should be banished from the standard library. Luckily, this banishment will likely never happen. And given the example code and benchmarks I'll show below, that's a good thing.

The two basic uses of stream IO for most games programmers are to do string formatting and to do file IO. We'll examine both of these and present benchmarks comparing stream IO methods to the "classic" methods of doing the same thing. C++ strings are given additional coverage in a later section. You will see that while I (and I *do* speak for Maxis here) am not a fan of C++ stream IO for game programming, I am a fan of the C++ string class itself.

C++ Stream IO for String Formatting

You don't see a lot of examples of string formatting in most C++ texts, because it is a relatively recent addition to the C++ standard library. Nevertheless, it has been around for a few years now and a brief description of it can be found in the excellent third edition of *The C++ Programming Language*, by Bjarne Stroustrup, section 21.5.3. Perhaps another reason you don't see a lot about the string stream IO system is that it is so painful to use, navigate, and understand.
Here are some examples of `stringstream` use:

`printf`-based string output formatting

```
#include <stdio.h>
char charArray[128];
sprintf(charArray, "The player %s has %d points.\n", "Bill", 320);
```

`ostringstream` equivalent:

```
#include <iostream>
using namespace std;
ostringstream ost;
ost << "The player " << "Bill" << " has " << 320 << " points.\n";
//You can now access the string stored in ost by calling ost.str().
```

The `ostringstream` output example above is harder to read than the `sprintf` example, and it runs about 25%-50% slower. It's likely to be harder to maintain as well, since interpreting what's going on is harder due to all the extra symbols. Actually, the `ostringstream` example above is about as clean as they get. Try doing number formatting with any kind of C++ stream and you'll want to give up programming altogether and take up something easier, like brain surgery. In preparing the example code for this section, I tried to find a way to clear out the above `ostringstream`. An hour later I gave up; perhaps it can't be done. That you can't do such a simple thing as this speaks for the ugly mess that C++ stream IO is. Now here's an example of `istringstream` input:

printf-based string input formatting:

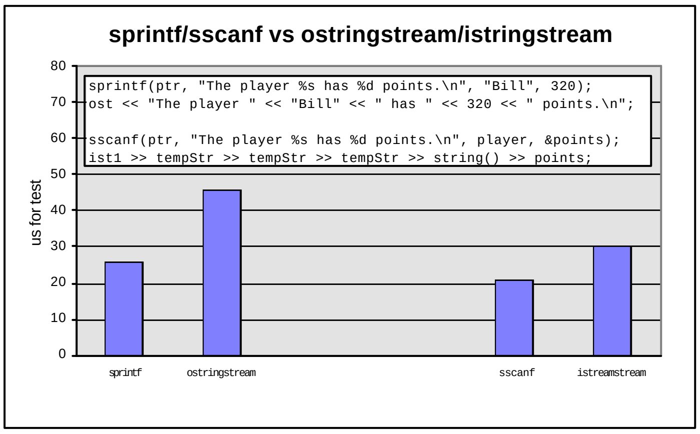
```
#include <stdio.h>
char charArray[128] = "The player Bill has 320 points.";
char player[32];
int  points;
sscanf(charArray, "The player %s has %d points.\n", player, &points);
```

istringstream equivalent:

```
#include <iostream>
using namespace std;
istringstream ist(string("The player Bill has 320 points."));
string        player, tempStr;
int           points;
ist >> tempStr >> tempStr >> player >> tempStr >> points;
```

Since C++ `istreams` want to read from objects, and not arbitrary data, the only built-in way to emulate the simple `sscanf()` call is to do what we did above. The example speaks for itself. It runs about 50% slower than the `sscanf()`. It is clear that I am trashing the design of the C++ streams here. While this may sound to some like merely personal opinion, empirically, a large majority of game programmers despise C++ streams. The above benchmarks and example both explain and justify this point of view. Nevertheless, C++ streams do have one advantage: IO can be defined for user-defined types. While `printf()` only knows about built in primitive types such as `int`, `char*`, and `float`, C++ streams can be overloaded with virtually any type. Game programmers rarely have a use for this kind of feature. In a later section we will inspect another novel feature of the C++ standard library, the STL, and come to a different conclusion.

Here is a graph of the code and results described above. While this is just one example, tests have shown that the results below are representative of the results you'll get with other cases.



```
sprintf(ptr, "The player %s has %d points.\n", "Bill", 320);
ost << "The player " << "Bill" << " has " << 320 << " points.\n";

sscanf(ptr, "The player %s has %d points.\n", player, &points);
ist1 >> tempStr >> tempStr >> tempStr >> string() >> points;
```

C++ stream IO for file operations

Here's an example of how to open a binary file and read in some data via C++ stream IO:

```
#include <fstream.h>
ifstream stream;
char     data[1024];
int      read_count;
stream.open("C:\\blah.dat", ios::in | ios::binary);
if(stream.is_open()){            //Note that open() returns void.
   stream.read(data, 1024);      //Note that read() doesn't return count
   read_count = stream.gcount(); //but instead returns stream&.
   stream.close();
}
```

You can also do C++ file IO with the streaming operators << and >>. Here is an example of how you would do this:

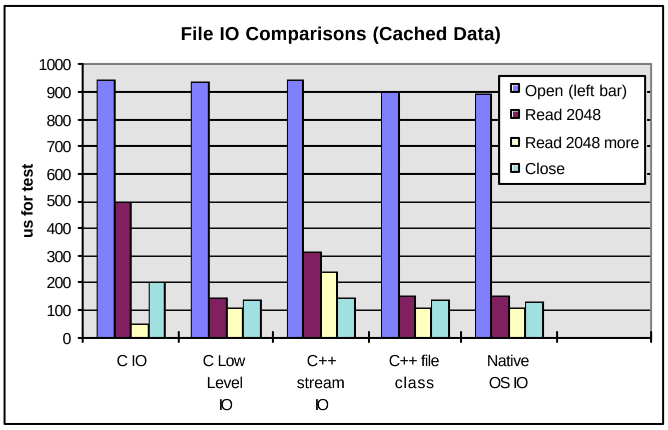
```
#include <fstream.h>

class A{
   long a;
   friend istream& operator>>(istream& is, A& a);
};

istream& operator>>(istream& is, A& a){
   is >> a.a;
   return is;
}

ifstream stream;
A        a;
int      read_count;
stream.open("C:\\blah.dat", ios::in | ios::binary);
if(stream.is_open()){            //Note that open() returns void.
   stream >> a;
   read_count = stream.gcount();
   stream.close();
}
```

C++ file IO via the streaming operators << and >> is not very friendly to the concept of reading in arbitrary binary data. The streaming operators need the operands to be classes, and those classes have to specifically have << and >> defined for them and the IO class. Yes, you can get around this, but to do so is messy and tedious. It's usually just easier to use `fstream::read()` and `fstream::write()` to do your work.

Let's get to the important part. How fast is the C++ stream IO compared to other methods? Here we do a simple comparison of five methods for opening a file, reading it, and closing it. When doing such comparisons, care needs to be taken that you account for file buffering that the operating system or run time library may be doing for you. Here are the five methods:

C IO	-- `fopen()`, `fread()`, ...
C low level IO	-- `_open()`, `_read()`, ...
C++ stream IO	-- `ifstream`
C++ home-grown file class IO	-- C++ wrapper around native OS IO
Native OS IO	-- Win32 `CreateFile()`

Here are some comparisons of cached file operations between the above-mentioned methods, after averaging many runs:



File opening took roughly the same time for all methods. File closing consistently took a little longer for C IO. All reading was slow for C++ stream IO, whereas the caching system of C IO caused first reads to be slow, but subsequent reads to be fast. C low level IO, C++ file class, and native OS all reads took basically the same amount of time. It is important to note that the results presented above represent the results of the using the C and C++ standard libraries of MSVC++ on Windows 95 and Windows NT. Other OSs and libraries may yield different results.

To my surprise, the C++ stream IO was consistently significantly slower than all other methods. Examining the source and disassembly of the various methods showed that the `ifstream` IO, under Visual C++, simply spent a lot of time doing housekeeping. The C IO and C low level IO methods basically spent a little time calling thread-safety functions, then called the native OS method to do the actual work. This explains why they were close in speed to the Native OS IO and home-grown C++ class IO. I recommend using a home-grown file interface for game programming, as it provides the most flexibility and speed, yet provides a platform-independent interface.

The example code for this section includes just such a class for Win32. It is implemented as `class MFile`, and is found in the files mfile.h/cpp (available at the web site and on the distribution CD). Here is an abbreviated version of the basic interface of MFile (there is a bit of extra functionality that's not shown below):

```
class MFile{
public:
MFile();
   MFile(const char* szPath);
   virtual ~MFile();

   virtual bool IsOpen() const;
   virtual bool Open(const char* szPath = NULL,
                     int openMode  = kOpenModeRead,
                     int openType  = kOpenTypeOpenExisting,
                     int shareMode = kShareModeRead);
   virtual bool Close();

   // Position manipulation
   virtual int    GetPosition() const;
   virtual int    GetLength() const;
   virtual bool   SetLength(int nNewLength);
   virtual int    SeekToBegin();
   virtual int    SeekToEnd();
   virtual int    SeekToRelativePosition(int nRelativePosition);
   virtual int    SeekToPosition(int nPosition);
   virtual int    Seek(int offset, SeekMethod origin);

   // I/O
   virtual bool Read          (void* buffer,       unsigned long  numBytes);
   virtual bool ReadWithCount (void* buffer,       unsigned long& numBytes);
   virtual bool Write         (const void* buffer, unsigned long  numBytes);
   virtual bool WriteWithCount(const void* buffer, unsigned long& numBytes);
   virtual bool Flush();
   ...
};
```

The take-home message here is that, unless the C++ stream IO classes are providing you with some extra necessary benefits, you probably want to stay away from them when writing high-performance code. Nowhere in the C++ language definition does it say you *must* use C++ stream IO. It is simply there if you want to use it. Like any other C++ feature, you can simply ignore it if you don't want to use it.

If you really want the highest-performance file IO, you might want to take advantage of two things: asynchronous IO and unbuffered IO. Asynchronous IO lets you start a file read in the background and let a callback function notify you when the read is complete. Unbuffered IO gives you fine control over exactly what bytes get read from the disk, making significant gains for random access file reads. I don’t have space to give examples here, but the methods are not too difficult under Win32 or MacOS. Other OSs may vary.

STL

STL stands for "Standard Template Library," and is the standard C++ method for implementing containers. It consists of the following templated classes:

Container	Description
`vector`	`Implements an array`
`list`	`Implements a doubly-linked list`
`queue`	`Implements a typical queue`
`deque`	`Implements a typical dequeue (double-ended queue)`
`priority_queue`	`Implements a queue with priorities assigned to each item`
`stack`	`Implements a standard stack`
`map`	`Implements a single-key associative container`
`multimap`	`Implements a multiple-key associative container`
`set`	`Implements a container of unique objects`
`multiset`	`Implements a container of non-unique objects`
`hash_set*`	`Implements a fast-access (but unsorted) set`
`hash_multiset*`	`Implements a fast-access (but unsorted) multi-set`
`hash_map*`	`Implements a fast-access (but unsorted) map`
`hash_multimap*`	`Implements a fast-access (but unsorted) multi-map`
`bitset`	`Implements a bit set, for compact data.`
`string`	`Implements a string class`
`valarray`	`Implements a vector specialized for numerics`

* As of this writing, the hash containers are not part of the C++ standard, though their interface and behaviour is well-defined enough that they can be considered a pseudo-standard. SGI provides a good implementation, but Microsoft does not. However, the has containers are entirely built upon the other standard containers, so in theory, you can use the SGI hash classes with the Microsoft version of the other STL classes.

As we will soon see, STL is a very viable library for C++ game programmers. Stroustrup describes the design like this: "The STL was the result of a single-minded search for uncompromisingly efficient and generic algorithms."[6] Virtually all the container classes listed above are useful to game programmers. We will examine some of the most common STL containers here and present a separate section on C++ strings afterward. STL classes are defined as C++ templates that heavily use inlining. In fact, the entire library is generally implemented entirely as header files. For STL to perform the way it was designed, it is critical that the compiler be good at implementing inlined functions and reasonably good at implementing templates. In fact, all current C++ compilers for both the PC and Mac are up to the task; our timing results will vouch for this. Console compilers are pretty good as well.

You may ask, "Does making templated inlined classes cause code bloat?" If you have 200 classes, and you declare STL containers for each of them, does it cause 200 copies of the entire STL class definition to be created and linked in? Basically, no. Since the classes are implemented as inlined templates with no virtual functions or multiple inheritance, any functions you don't call will never get used. The functions that do get used will be implemented inline. While this is certainly faster than implementing them as separate functions, code that gets called repeatedly effectively gets instantiated multiple times. However, STL classes are generally small classes whose most common operations can be implemented with just a few instructions. So in practice, while there is a slight size increase, practice has shown that it is not very much. A common reply of fans of basic C containers is that you can simply create a single C list struct that holds pointers to `void*` and thus you can store virtually anything in it. Well, you can do this same thing with an STL container as well, and get identical performance, so it is a non-issue. See the **Templates** section for more information about templates–in particular, issues with finding syntax errors in code that uses templates.

Lastly, there is the issue of allocators. For every container type, STL allows you to supply your own custom allocator. As game programmers know well, `malloc`/`new` and `free`/`delete` are very slow compared to custom fixed-size allocators. STL lets you supply your own allocator to the container, resulting in the fastest possible code. I don't have space here to give examples of this, but any decent C++ reference will cover it.

vector

The `vector` class implements an array. While you can certainly use the standard built-in array mechanism of the language, the `vector` class offers some extra features with little cost. The class is designed to be syntactically just like the language's built-in array. It achieves this by defining an inline `operator[]` for the class. Here is a side-by-side comparison between the two array types:

Built-in array	STL vector
`int intArray[300];`	`vector<int> intVector(300);`
`intArray[3] = 5;`	`intVector[3] = 5;`
`x   = intArray[3];`	`x   = intVector[3];`
`ptr = intArray+3;`	`ptr = & intVector[3];`

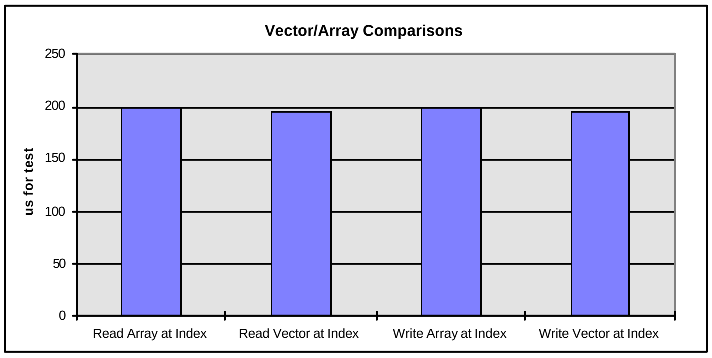
Notice that the syntax is almost identical. Note that with an STL vector, you must access individual elements via `operator[]`, and can't use syntax like "`intArray+3`", as shown above. This isn't a problem, because accessing the array through `operator[]` is *exactly* as fast as the "`intArray+3`" method. With an STL vector, you get extra features as well. Here are some examples of other functions you can use with `vector`:

```
count = intVector.size();                //Get the current size.
intVector.push_back(7);                  //Adds an entry. Resizes if necessary.
intVector.reserve(400);                  //Pre-allocate space for 400 items.
y = intVector.size();                    //How much is stored?
y = intVector.capacity();                //How much is reserved? Greater that or equal
                                           to how much is stored.
intVector.insert(ptr,1,5);               //Insert one instance of value "5" at the
                                           given position.
```

Here I present some timings and disassembly comparing STL `vector` access to the built-in array mechanism. As with all our examples, the self-timing sample code is available on the distribution CD and the HPGP web site. First we compare disassembly of the two methods:

```
39:  intArray[3] = 5;
     mov dword ptr [ebp-3A88h],0005h

48:  intVector[3] = 5;
     mov dword ptr [esi+5C0h],0005h
```

They are identical. Here is a timing comparison of a series of accesses to the arrays. Arrays and Vectors are identical:



list

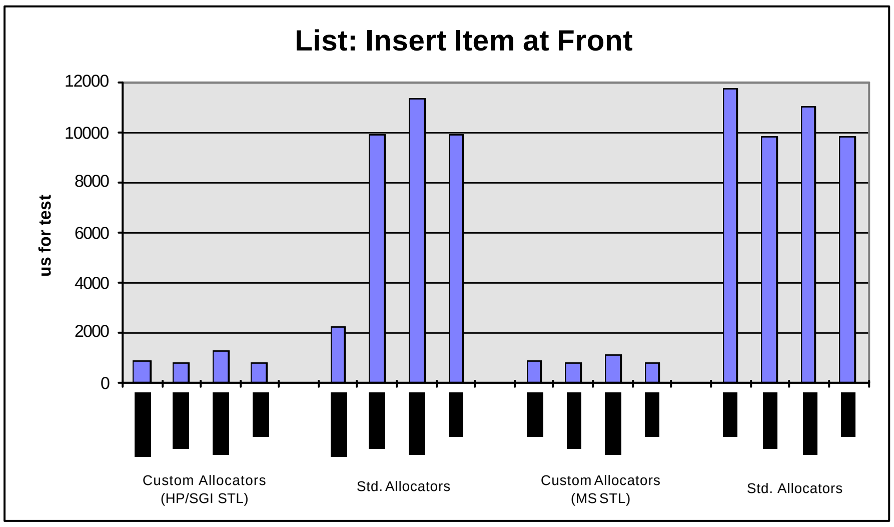
The `list` class implements a generic doubly-linked list. As we will see, it is as fast as any generic C or C++ doubly-linked list can be. Here is a partial example of `list` functionality:

```
list<int> intList;            //Create the list.
list<int>::iterator it;       //Create a list iterator.
intList.push_back(5);         //Add an entry to end.
intList.pop_front();          //Remove from front.
intList.push_front(7);        //Add to front.
intList.pop_back();           //Remove from end.
x = intList.size();           //Get count of elements.
intList.merge(intList2);      //Merge two lists.
it = find(intList.begin(),    //Find an entry of value
          intList.end(), 5);  //   equal to 5.
intList.insert(it, 6);        //insert 6 in front of 5.
intList.sort();               //Sort the list.
```

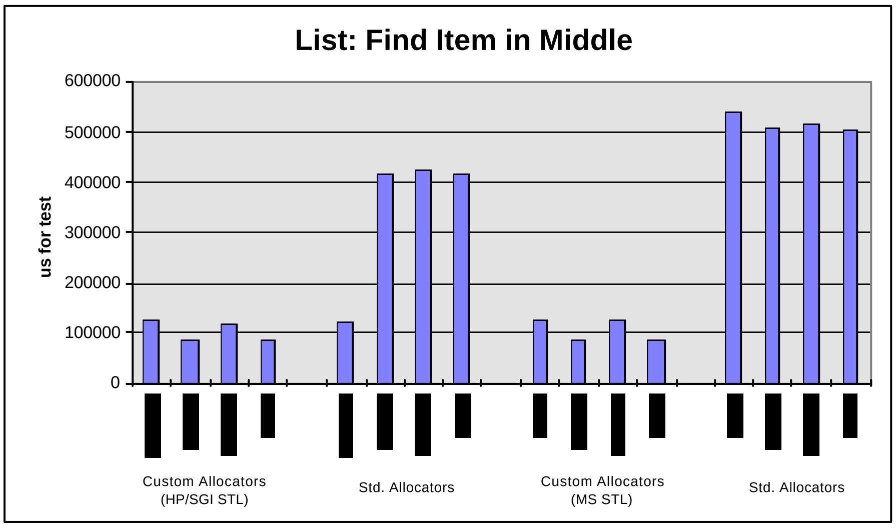
Here I present benchmarks comparing four types of lists:

A hand-coded singly-linked list in C
A hand-coded singly-linked list class in C++
A hand-coded doubly-linked list class in C++
The STL list template class (Microsoft and HP/SGI versions)

First we compare the time needed to insert an item at the front of the list. After examining the results below, one thing may strike you. HP/SGI STL yields very fast times when the standard allocator is used (fifth bar from the left). The reason this is happening is that HP/SGI STL implements a fast allocator even if you don't supply one. The basic result here is that insertion in the STL list container is as fast as with a generic doubly-linked list and nearly as fast as a generic singly-linked list. Singly-linked lists have a slight speed advantage (~5%), but have the disadvantage of being less flexible. For some things, you can get by with the singly-linked list.



Next we compare the times needed to find an item in the list. You might think that they'd all take the same time, but they don't. When custom allocators are used, the time to search goes way down. What's happening? The answer is that the custom allocator keeps the items much closer together in memory, using the processor cache (L1, and possibly L2) more efficiently. If you weren't previously convinced of the difference the cache can make, you should be now.



I don't have space to deal with most of the other STL classes here, but if you do your own investigations, you'll see that they too are very efficient. In particular, you'll find the `map`, `set`, `hash`, and `string` classes to be very useful. We will take a little time now to examine the `string` class.

C++ Strings

C stands out as one of the few programming languages that doesn't have native support for strings. Instead, C takes the view that an array of characters is the same thing as a string and leaves it up to the compiler vendor to provide a standard library of functions (such as `strlen()`) that operate on these character arrays. Despite the fact that C takes pride as an efficient language, C strings are one of the least efficient parts of the language, largely due to the design that a string in C is simply a `NULL`-terminated array of characters. Why is this inefficient? Two reasons.

First, since the length of the string is defined by where the first `NULL` character is, the only way to tell the length of the string is to read the characters until you run into the `NULL`. And if you want to append one string to another, you have to know the length of the destination string. Now you may argue that storing the length of the string would require an overhead of extra bytes that the original designers of C couldn't afford to waste. That's fine. I'm not concerned here with criticizing the original C architects, I'm simply explaining the state of things as they are now.

Secondly, since the compiler treats strings like any other array, it can't make optimizations that might make string processing faster, such as padding the end in anticipation of concatenation and other length-changing operations.

The C++ `string` class attempts to rectify the problems found in C strings. Here is a partial listing of the string class, simplified for readability.

```
class string{
public:
   string(char* ptrToNullTerminatedCharArray);
   string(char* ptrToData, size_t lengthOfData);
   size_t  length();
   void    resize(size_t length, char ch);
   char*   c_str();
   size_t  capacity();
   string& operator +=(char ch);
   string& operator +=(char* ptrToNullTerminatedCharArray);
   string& operator +=(const string& srcString);
   string& operator = (const string& srcString);
   char&   operator[];
   size_t  find(char ch, size_t position=0);
   size_t  find_first_of(const string& str, size_t position=0) const;
};
```

The actual C++ `string` class is a template, which allows strings based on both `char` and `wchar_t` (wide characters) to have identical interfaces. Also, there are a number of other functions present in addition to the ones shown above.

Here are some of the advantages of the C++ string class over C strings:

Length is always known, so length-changing operations are efficient.
Since C++ strings don’t use `NULL` termination, the `NULL` character itself is a valid independent character in a C++ string.
Nearly transparent support for wide character strings.
Faster than C strings for most situations.

Here are some disadvantages of the C++ `string` class:

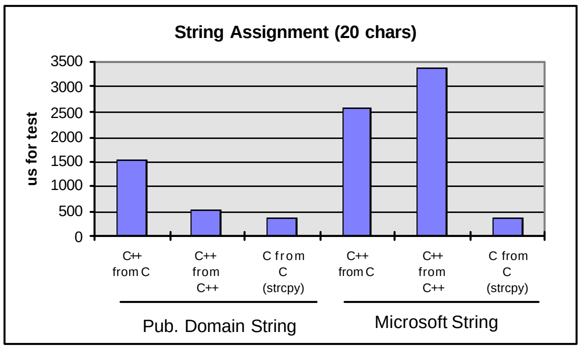
Operating systems can’t know about the C++ `string` class.
`c_str()` may be inefficient, depending on which STL you use.

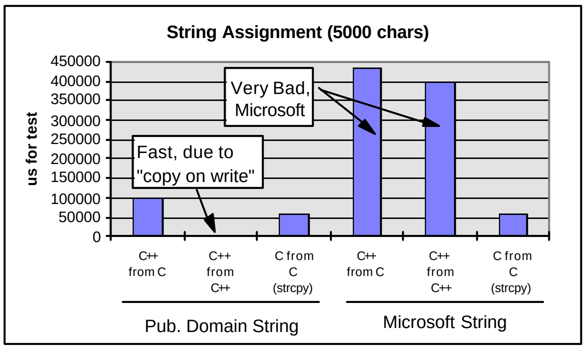
If you have a C++ `string` and need to get a null-terminated C string from it, you must call the `string` class member function `c_str()`. What `c_str()` does is return a pointer to the `string` object as a null-terminated character array. Note that this may or may not be equivalent to simply getting the address of the first character (`&string[0]`) or calling data. Some older versions (e.g. the old Modena "`bstring`" implementation), but no common current versions, of the C++ standard library have a separate character array pointer used for `c_str()`. Their `string` class is effectively written like this:

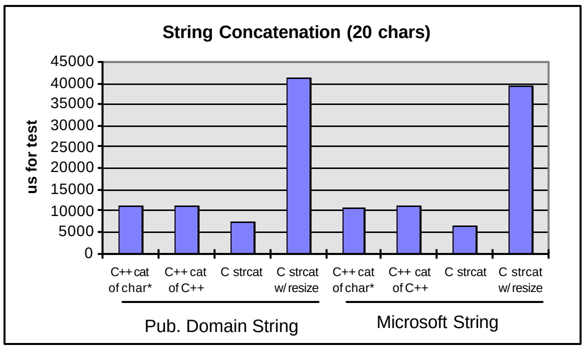
```
class string{
   …
   char* data;
   char* data_for_cstr;

   const char* c_str(){   // *very* slow function.
      delete data_for_cstr;
      data_for_cstr = new char[length()];
      strncpy(data_for_cstr, data, length());
   }
   …
};
```

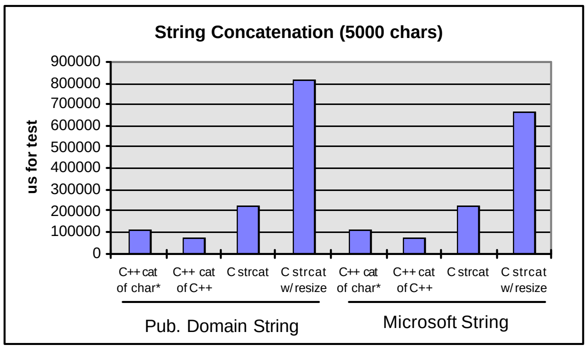
Most current `string` class implementations declare only one data pointer, and it is kept in `NULL`-terminated form during operations, allowing the `c_str()` function to be an inline that simply returns the data pointer itself.

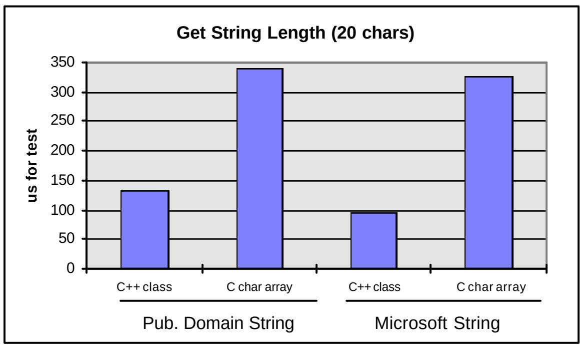
So how fast are `string` operations? In reality, the results of benchmarking strings depends a lot on the contents of the strings. Results could vary by up to a magnitude or more. I could simply say this and go on to the next topic, confident that I gave you the correct academic response. But that would, as such things always are, be unsatisfying. The fact is that we *can* set up some benchmarks that give you a good sense of how fast C++ string operations are, at least with respect to the equivalent C operations. When reading these results, think about the general result, and not the exact numbers or percentages. Those general results are what you will see in your own code.

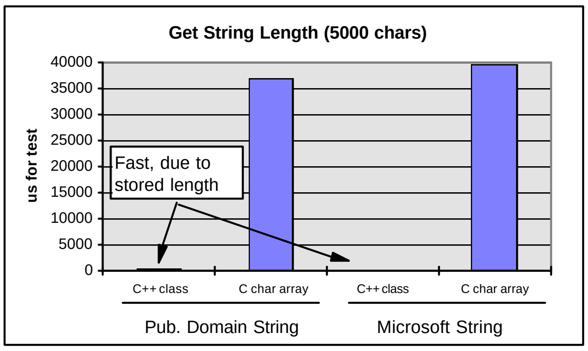












String assignment with C++ strings is somewhat dependent on whether the C++ string has to resize itself and whether the class implements “copy on write.” Microsoft’s string implementation is bad on both counts: it resizes itself on every assignment, even if it doesn’t need to, and it doesn’t implement “copy on write.” Other string tests, such as substring searching, `printf` formatting, and string comparing were done. I don’t have space for the results here, but the results were as expected: C++ strings usually outperformed C char array strings when the C++ string could take advantage of its knowledge of itself or the other string in the expression. C `char` array strings usually outperformed C++ strings when the operation forced a reallocation of the C++ string data pointer.

Templates

Templates are an "abstraction mechanism" which allows you to define classes and functions that can be bound to types separately. We will assume that you understand them already, because it would take up too much space to describe them here.

Templates are pretty useful at times for game programmers. Here's an example of a useful templated class:

```
template <class T>
struct RectTemplate{
   T left;
   T top;
   T right;
   T bottom;

   Rect(){}
   Rect(const T& newL, const T& newT, constT& newR, const T& newB);
   T Width();
   T Height();
   bool PointInRect(const T& x, const T& y);
   void Inflate(const T& horizontal, const T& vertical);
};
```

This is very useful because sometimes you want a rectangle class that uses four byte `ints`, but other times you need to be very thrifty and want to make a rect that uses one byte `chars`. With templates, you need only define and debug the class once. Here's how you make and use the two `Rect` types mentioned above:

```
typedef RectTemplate<int>  IntRect;  //Declare a rect that uses ints.
typedef RectTemplate<char> CharRect; //Declare a rect that uses chars.

IntRect  intRect(0,0,15,13);   //Declare an IntRect instance.
CharRect charRect(0,0,15,13);  //Declare a CharRect instance.
intRect.Inflate(3,2);    //Use the IntRect.
charRect.Inflate(3,2);   //Use the CharRect.
```

Now `IntRect` and `CharRect` are full-fledged classes, just as if you had written them independently. This is cool. But how efficient is it? Are these templated classes as fast as if you wrote them as regular standalone classes? How much memory do they use? I partially answered these questions in the STL section earlier. If you will recall, the basic answer is that they are *exactly* as fast as if you wrote them as standalone classes. How much space do they take up? In the case of STL, the entire library is implemented inline, so if the compiler is set to actually inline everything it sees, STL classes will indeed increase the size of the app, though likely improve the speed at the same time.

One key to understanding how templates work is to pretend that every time you instantiate a templated class (e.g. when I declared `IntRect` above), pretend that you take the header file for the class, save it as another file, and do a search and replace, replacing every "T" with your type. Pretend that this new file is what actually gets compiled. In fact this is more or less conceptually how compilers actually implement templates. If you understand this, you can automatically answer most technical questions about what kind of syntax is legal, what is efficient, why some piece of code won't compile, and so on.

One unfortunate problem with templated classes is that some compiler error printouts with templated classes are practically unreadable. Take this STL template compiler error generated by MSVC++, for example:

```
46: struct Blah{};
47: list<Blah> blahList;
48: blahList.insert(blahList.begin(), 3);
```

```
main.cpp(48) error C2664: 'class list<struct Blah>::iterator list<struct
Blah>::insert(class list<struct Blah>::iterator, const struct Blah &)' : cannot convert
parameter 2 from 'const int' to 'const struct Blah &'
```

Deciphering this may take you 10-20 seconds. But if you read it through, you can figure it out. Usually, template compiler errors for your own templated classes are much more straightforward than those related to STL, due to the syntactical complexity of STL implementations.

There is another issue with writing code that uses templates, in particular, STL templates. When some compilers (Microsoft, in particular) encounter an error in code that uses STL, the compiler will tell you that some STL file and line is in error. Well, we are pretty sure that the STL implementation is debugged, so why is it telling us the code is in error? The answer is that STL containers often expect the classes that they contain to have certain properties, such as having an `operator==()` defined for it. If you don't have this operator, STL code will not compile, because the line in error is using the == operator as if it existed. This is really not a template problem, but a problem than can happen with any C or C++ code. But the problem tends to come up with templates a lot. So be prepared for this.

So what's the take-home message on templates? Templates are very useful, are just as fast as regular classes, may increase the size of your code, and are pretty easy to use, except sometimes when debugging usage syntax. Templates have been around for years. Most current C++ implementations, and all C++ implementations for microcomputers that I've seen, have them. So you can feel pretty safe using them.

Conclusions

We've examined the most common features of C++, examined what happens under the hood, and measured and compared the mechanisms. What we've learned is that if you understand how C++ works, you can tell the efficient features from the inefficient ones. While some parts of the language appear to be more academic than practical, most parts are simple, efficient, useful additions that benefit game programmers as much as anyone. And remember, when writing high-performance code with any language, *always* check the disassembly and *always* time your code.

Credits
I would like to thank the following people for help in the review and editing of this document:
Kevin O'Hare, Laura O'Shea, Eric Bowman, and Jason Shankel.

[1] GNU C++ FAQ: http://www.cs.ucr.edu/docs/texinfo/g++FAQ/g++FAQ_toc.html
[2] Effective C++: 50 Specific Ways to Improve Your Programs and Designs, by Scott Meyers
More Effective C++ : 35 New Ways to Improve Your Programs and Designs, by Scott Meyers
[3] Zen of Assembly, by Michael Abrash.
[4] Pentium Processor Optimization Tools, by Michael Schmit, published by AP Professional, 1995
See also:
Inner Loops. Sourcebook for Fast 32 Bit Software Development, by Rick Booth, published by Addison Wesley.
[5] A Crash Course on the Depths of Win32 Structured Exception Handling, by Matt Pietrek. MSJ, Vol 12, #1, Jan. 1997
[6] The C++ Programming Language, by Bjarne Stroustrup. Third edition, Section 16.2.3