Writing a Kernel in C++

Tools

Examples will show how to use :-

- **DJGPP** - a complete 32-bit C/C++ development system for Intel 80386(and higher) PCs running DOS.
- **NASM** - The Netwide Assembler Project - Open sourced 80x86 assembler

Source code listed on this site has only been tested on djgpp version 2.03 and nasm version 0.98.08, running on the Window XP environment. Some of the techniques discussed here will not work on other versions of gcc. I will try to give a warning for compiler specific implementations.

Assumptions

I assume that this is not your first attempt at writing a Kernel.

I assume you are already proficient in the programming language C++.

I assume you have already written a boot loader, or that you know how to use a standard one like **GRUB**. If you do use grub, you will need to add the necessary code to the assembler code below.

Part 1: Introduction, "Hello, world!" kernel - C++ with no Run-Time support

Introduction

The aim of these tutorials is to show you how to implement a simple kernel written in C++. We will need a small amount of assembler (to initialise our kernel), and a smidgen of C code (to provide some C++ runtime support). The end result, will be the typical "Hello, world!" example.

Part 1, will introduce the Video driver, with no Run-Time support.
Part 2, will help add some run-time support, enabling global / static objects.
Part 3, will demonstrate a very simple implementation of std::cout (Standard output stream in the C++ Standard lib).

The end result will be this (*taken from the first example in Accelerated C++, Andrew Koenig and Barbara E. Moo*) :-

```
//a small C++ kernel
#include <iostream>

int main()
{
   std::cout << "Hello, world!" << std::endl ;
   return 0 ;
}
```

Part 4, will set-up a global descriptor table, and an interrupt descriptor table. With some encapsulation in C++.
Part 5, will introduce a simple interrupt driven Keyboard driver.
Part 6, will demonstrate a very simple implementation of std::string
Part 7, will demonstrate a very simple implementation of std::cin.

The end result will be this (*taken from the second example in Accelerated C++, Andrew Koenig and Barbara E. Moo*) :-

```
//a small C++ kernel
#include <iostream>
#include <string>

int main()
{
   //ask for the person's name
   std::cout << "Please enter your first name: " ;
   //read name
   std::string name ;       //define name
   std::cin >> name ;      //read into name

   //write a greeting
   std::cout << "Hello, " << name << "!" << std::endl ;
   return 0 ;
}
```

As you can see, there is more work involved to get the C++ kernel up and running, at least compared to a C kernel, but hopefully the end result will be a solid base onto which to build a more flexible object oriented kernel.

Knowing our limits

Quoting from the creator of C++, Bjarne Stroustrup's book "The C++ programming language third edition (section 1.3.1)",

> **"Except for the *new*, *delete*, *typeid*, *dynamic_cast*, and *throw* operators and the *try-block*, individual C++ expressions and statements need no run-time support."**

The features which need Run-Time support full into 3 categories :-

- Built in functions (new, delete),
- Run Time Type Information (typeid, dynamic_cast),
- And Exception handling (throw, try-block).

However, Stroustrup was referring to C++ code on an operating system which has an implementation of the C++ standard library. So this must also be implemented or ported for use in the kernel.

The good news is we can disable these features with most compilers, and everything will be ok as long as we don't use those off-limit expressions and statements. When the time comes when we want to use them, we have to add our own support code and link it to our kernel.

```
example of how to disable these in gxx, the win32 g++
compiler.
gxx -c *.cpp -ffreestanding -nostdlib -fno-builtin -fno-rtti
-fno-exceptions

offsite link : Options Controlling C++ Dialect
```

The bad news is there is no standard way that built in functions, RTTI, or EH have been implemented into the compiler. **Even different versions of the same compiler may do things differently**. In part 2, I'll explain in more detail how we can tackle this problem, but for now we will just disable them all.

When we disable the run-time support in the compiler, the compiler will omit several important functions. The compiler normally makes a call to a function before calling main(), and another after main() returns. Typically these two functions will be called _main() and _atexit(). Amongst other operations, they normally handle the calling of global / static objects constructors and deconstructors. So global / static objects are also off-limits until we add the necessary support code for this.

To summarise, a list of the off-limit C++ features (without adding your own support code) :-

- Built in functions,
- Run Time Type Information,
- Exception handling,
- The C++ standard library (including the C library of course),
- And global / static objects.

The code

Lets start with the ASM code which will call our kernel's main function. Later this code will also make calls to our run-time support, _main() and _atexit().

```
; Loader.asm

[BITS 32] ; protected mode

[global start]
[extern _main] ; this is in our C++ code

start:
call _main ; call int main(void) from our C++ code
cli ; interrupts could disturb the halt
hlt ; halt the CPU
```

Now in our C++ Kernel, we are going to create a class Video, which will act as a simple video driver. This is the entire code for the kernel.

```
//Video.h
```

```
#ifndef VIDEO_H
#define VIDEO_H       //so we don't get multiple definitions of
Video

class Video
{
public:
      Video() ;
      ~Video() ;
      void clear() ;
      void write(char *cp) ;
      void put(char c) ;
private:
      unsigned short *videomem ;   //pointer to video memory
      unsigned int off ;                  //offset, used like a y cord
      unsigned int pos ;                 //position, used like x cord

}; //don't forget the semicolon!

#endif



//Video.cpp
#include "Video.h"

Video::Video()
{
      pos=0 ;       off=0 ;
      videomem = (unsigned short*) 0xb8000 ;
}

Video::~Video() {}

void Video::clear()
{
      unsigned int i;

      for(i=0; i<(80*25); i++)
      {
            videomem[i] = (unsigned char) ' ' | 0x0700 ;
      }
      pos=0 ; off=0 ;
}

void Video::write(char *cp)
{
      char *str = cp, *ch;

      for (ch = str; *ch; ch++)
      {
            put(*ch) ;
      }
```

```
}

void Video::put(char c)
{
	if(pos>=80)
	{
		pos=0 ;
		off += 80 ;
	}

	if(off>=(80*25))
	{
		clear() ; //should scroll the screen, but for now, just
clear
	}

	videomem[off + pos] = (unsigned char) c | 0x0700 ;
	pos++ ;
}
```

```
//Kernel.cpp
#include "Video.h"

int main(void)
{
	Video vid ;  //local, (global variables need some Run-Time
support code)

	vid.write("Hello, world!") ;
}
```

Compiling

Video.cpp and Kernel.cpp need to be compiled with a C++ compiler, remembering to disable the above mentioned C++ features. The output from your C++ compiler should be the object files Video.o and Kernel.o. Loader.asm also needs to be assembled with an assembler. The output from your assembler should be the object file Loader.o.

An example of how to compile using DJGPP gxx and NASM.

```
gxx -c Video.cpp -ffreestanding -nostdlib -fno-builtin -fno-rtti
-fno-exceptions
gxx -c Kernel.cpp -ffreestanding -nostdlib -fno-builtin -fno-rtti
-fno-exceptions
nasm -f aout Loader.asm -o Loader.o
```

Linking

Now we must link our object files into a flat binary which we shall call Kernel.bin.

I recommend using a linker script, we will be making use of the linker script in part 2. Here is an example of a linker script for LD.

```
/* Link.ld */
OUTPUT_FORMAT("binary")
ENTRY(start)
SECTIONS
{
   .text 0x100000 :
   {
      code = .; _code = .; __code = .;
      *(.text)
      . = ALIGN(4096);
   }

   .data :
   {
      data = .; _data = .; __data = .;
      *(.data)
      . = ALIGN(4096);
   }

   .bss :
   {
      bss = .; _bss = .; __bss = .;
      *(.bss)
      . = ALIGN(4096);
   }

   end = .; _end = .; __end = .;
}
```

Now we can use the linker script with LD,

```
ld -T Link.ld -o Kernel.bin Loader.o Kernel.o Video.o
```

Conclusion

Hopefully your C++ Kernel should have compiled and linked without any errors. Congratulations.

Part 2: Introduction, "Hello, world!" kernel - C++ with Global / Static Object support

::Warning - Compiler Specific::

Solution 1 - .ctor and .dtor sections

This method will only work for compilers which add two sections to the object files, the .ctor and .dtor section. Here are 4 steps to find out if you can use this solution :-

Step 1 - Move the Video object from a local to global scope.

```
//Kernel.cpp
#include "Video.h"

Video vid ;  //global variable

int main(void)
{
      vid.write("Hello, world!") ;
}
```

Step 2 - Compile Kernel.cpp for a freestanding environment, without run-time support, or standard library.

```
gxx -c Kernel.cpp -ffreestanding -nostdlib -fno-builtin -fno-rtti
-fno-exceptions
```

Step 3 - Use the object dump tool to display contents of the sections.

```
objdump -h Kernel.o > Kernel.dis
Redirects the output of objdump to a file named Kernel.dis
```

Step 4 - Open the Kernel.dis with a text editor (Notepad, WordPad, my preferred choice is TextPad)

```
Look for something like this in the file
7 .ctors 00000004 000000f4 000000f4 00000294 2**2
CONTENTS, ALLOC, LOAD, RELOC, DATA
8 .dtors 00000004 000000f8 000000f8 00000298 2**2
CONTENTS, ALLOC, LOAD, RELOC, DATA.
If they are present, you can use solution 1.
```

The code

We are now going to implement _main() and _atexit(). In our linker script we are going to create a list of pointers in the .ctor section, and a list of pointers in the .dtor section. Which are the Constructor lists, and Deconstructor lists respectively.

Note - The list is not sorted into order of precedence, in a complex hierarchy the constructors could be called in an incorrect order!

```
//Support.c
void _main()
{
   //Walk and call the constructors in the ctor_list

   //the ctor list is defined in the linker script
   extern void (*_CTOR_LIST__)() ;

   //hold current constructor in list
   void (**constructor)() = &_CTOR_LIST__ ;

   //the first int is the number of constructors
   int total = *(int *)constructor ;

   //increment to first constructor
   constructor++ ;

   while(total)
   {
      (*constructor)() ;
      total-- ;
      constructor++ ;
   }
}

void _atexit()
{
   //Walk and call the deconstructors in the dtor_list

   //the dtor list is defined in the linker script
   extern void (*_DTOR_LIST__)() ;

   //hold current deconstructor in list
   void (**deconstructor)() = &_DTOR_LIST__ ;

   //the first int is the number of deconstructors
   int total = *(int *)deconstructor ;

   //increment to first deconstructor
   deconstructor++ ;

   while(total)
   {
      (*deconstructor)() ;
      total-- ;
      deconstructor++ ;
   }
}
```

```
; Loader.asm

[BITS 32] ; protected mode

[global start]
[extern _main]    ; this is in our C++ code
[extern __main]   ; this is in our C support code
[extern __atexit] ; this is in our C support code

start:
call __main
call _main ; call int main(void) from our C++ code
call __atexit
cli ; interrupts could disturb the halt
hlt ; halt the CPU
```

```
/* Link.ld */
OUTPUT_FORMAT("binary")
ENTRY(start)
SECTIONS
{
   .text 0x100000 :
   {
      code = .; _code = .; __code = .;
      *(.text)
      . = ALIGN(4096);
   }

   .data :
   {
      __CTOR_LIST__ = .; LONG((__CTOR_END__ -
__CTOR_LIST__) / 4 - 2) *(.ctors)   LONG(0) __CTOR_END__
= .;

      __DTOR_LIST__ = .; LONG((__DTOR_END__ -
__DTOR_LIST__) / 4 - 2) *(.dtors) LONG(0) __DTOR_END__ =
.;

      data = .; _data = .; __data = .;
      *(.data)
      . = ALIGN(4096);
   }

   .bss :
   {
      bss = .; _bss = .; __bss = .;
      *(.bss)
      . = ALIGN(4096);
```

```
    }

    end = .; _end = .; __end = .;
}
```

Conclusion

Compile and link as in Part 1. You will now be able to use Global / Static objects in your C++ Kernel. In Part 3 we will look at a simple implementation of Class OStream which resides in namespace std. We will use a global instance of OStream, cout.