UEFI APPLICATIONS WITH C++

Morris Hafner (@mmhafner)

ABOUT ME

- Moved from Germany to Scotland
- Software engineer at Codeplay Software
- Postgrad student at University of Edinburgh
- Tendency to break compilers

WHAT IS (U)EFI?

- “New” firmware interface for x86, ARM, RISC-V
- Many features compared to BIOS
    - Network, Secure Boot, NVRAM, EFI Bytecode, ...
- No assembly required for many applications (e. g. bootloaders)

THE OLD WAYS: BIOS

- Loads 512B from MBR and jumps to 0x7C00
- Requires assembly code
- Bootloader switches to Protected Mode (32 Bit) and Long Mode (64 Bit)
- Bootloader scans ACPI, PCI, SMBus…
- May invoke BIOS services via interrupts

EFI EXECUTABLES

- COFF file format, subsystems 10-13
- MS calling convention
- UTF-16 strings
- On a FAT32 partition with partition type `0xEF00`
- Implementations usually default to executing `EFI/Boot/bootx64.efi`

FIRST EXECUTABLE

TOOLCHAIN

- TianoCore EDK II, GNU efilib
- We will use GNU efilib, it is easier to use
- Available compilers: MSVC, MinGW gcc
- No clang
    - Doesn’t support freestanding COFF executables

STANDARD LIBRARY

- no C or C++ standard library
  - A C95 stdlib implementation is part of EDK II
- Heap not easily accessible
- In practice, many headers can still be used
  - most of `<algorithm>`, `<array>`,`<tuple>`, `<type_traits>`

COMPILER INVOCATION

```
x86_64-w64-mingw32-g++ \
  -mno-red-zone \
  -ffreestanding -fshort-wchar \
  -nostdlib -e efi_main \
  -Wl,-dll -shared -Wl,--subsystem,10 \
  -c main.cpp
```

This can be wrapped in a CMake Toolchain

THE CODE

```
#include <efi.h>
#include <efilib.h>

extern "C" [[gnu::ms_abi]]
EFI_STATUS efi_main(
  IN EFI_HANDLE ImageHandle,
  IN EFI_SYSTEM_TABLE *SystemTable)
{
    SystemTable->ConOut->OutputString(
      SystemTable->ConOut,
      (CHAR16 *) L"Hello World\r\n"); // Note the missing const
}
```

RUNNING IT

OVMF is an OSS implementation that can be used with QEMU

```
qemu-system-x86_64 \
-drive file=hdd.img,if=ide \
-bios OVMF.fd
```

ACCESSING EFI SERVICES

THE EFI PROTOCOL INTERFACE

- Firmware services can be queried using GUIDs
- Everything is loaded into structs
- OO style interface, but in C with function pointers

```
EFI_STATUS
LocateDevicePath (
 IN     EFI_GUID                 *Protocol,
 IN OUT EFI_DEVICE_PATH_PROTOCOL **DevicePath,
 OUT    EFI_HANDLE               *Device
 );
```

HMMM...

Signature is usually

```
EFI_STATUS(in..., inout *..., out *...)
```

In C++, we want this:

```
expected<tuple<out...>, EFI_STATUS> const res =
  func(in..., inout &...);
if(res) {
   auto[out...] = res.value();
}
```

WHAT TO DO?

```
template <size_t in_count, size_t inout_count,
          size_t out_count, typename Callable>
auto wrap(Callable c);

auto locate_device_path = wrap<1, 1, 1>(LocateDevicePath);
expected<tuple<EFI_HANDLE>, EFI_STATUS> result =
  locate_device_path(protocol, device_path);
```

1. Partition the argument list
2. Create a new function wrapping the EFI function
3. Wrap the error codes

GETTING THE ARGUMENT LIST

→ Boost Callable Traits

```
EFI_STATUS
LocateDevicePath(EFI_GUID *,
  EFI_DEVICE_PATH_PROTOCOL **,
  EFI_HANDLE *);

#include <boost/callable_traits/args.hpp>
using args =
  boost::callable_traits::args_t<decltype(LocateDevicePath)>;
static_assert(is_same_v<args,
  tuple<EFI_GUID *,
        EFI_DEVICE_PATH_PROTOCOL **,
        EFI_HANDLE *>
);
```

PARTITION THE ARGUMENT LIST

```
template <size_t offset, typename T, size_t... I>
auto constexpr st_impl(T tuple, index_sequence<I...>) {
    return tuple{get<offset + I>(tuple)...};
}

template <size_t N, typename T>
auto constexpr split_tuple(T tuple) {
  auto constexpr sz = tuple_size_v<T>;
  return tuple{
    st_impl<0>(tuple, make_index_sequence<N>{}),
    st_impl<N>(tuple, make_index_sequence<sz - N>{}),
  };
}
```

3-WAY SPLIT

```
using in_split =
  decltype(split_tuple<in_count>(declval<args>()));
using In = // <- first
  tuple_element_t<0, in_split>;
using InOut_Out =
  tuple_element_t<1, in_split>;
using inout_out_split =
  decltype(split_tuple<inout_count>(declval<InOut_Out>()));
using InOut = // <- second
  tuple_element_t<0, inout_out_split>;
using Out = // <- third
  tuple_element_t<1, inout_out_split>;
```

WRAP THE FUNCTION

```
template <typename Func,
  typename... Is, typename... IOs, typename... Os>
auto constexpr make_out_param_adapter(
  Func func, tuple<Is...>, tuple<IOs...>, tuple<Os...>) {
    return [func](Is... is, IOs... ios) {
      tuple<remove_pointer_t<Os>...> res;
      auto const ptr = transform_tuple(res, [](auto &val) {
        return addressof(val);
      });
      apply(c, tuple_cat(
        tuple{get_ptr(is)...}, tuple{get_ptr(ios)...}, ptr));
      return res;
    };
}
```

OVERHEAD?

OVERHEAD!

`std::tuple` causes value initialization

```
template <typename T> struct uninitialized {
  uninitialized() {}
  T & get() {
    return val;
  }
  T val;
};
```

```
tuple<uninitialized<remove_pointer_t<Os>>...> res;
auto const ptr = transform_tuple(res, [](auto &val) {
  return val.get();
});
```

WRAP THE ERROR CODES

- `expected<T, E>` contains either a value or an error
- We use Simon Brands `tl::expected<T, E>`

```
if constexpr(
  is_same_v<invoke_result_t<Func, Is..., IOs..., Os...>,
            EFI_STATUS>) {
  using result_t =
    tl::expected<tuple<remove_pointer_t<Os>...>, EFI_STATUS>;
  if(auto result = apply(...) != EFI_SUCCESS)
    return result_t(tl::unexpect, result);
  return result_t{transform_tuple(...)};
} else { // void
  apply(...);
  return res;
}
```

OVERHEAD?

WE MADE SIMPLIFICATIONS

- Assumptions that there are only fundamental types and PODs
  - Types must be trivially constructible (`uninitialized<T>`)
  - Compiler can go much further thanks to trival copy construction and destruction
- No overloads (Callable Traits)
  - Not too bad, C doesn’t have overloads or destructors
- But may be wrapped by a future `std::overload`

EXAMPLE APPLICATION

RENDERING TO THE FRAMEBUFFER

- We could write our own kernel in C++ now...
  - `ExitBootServices()`
- Let’s render a couple of spheres onto the framebuffer instead

CREATING A GRAPHICS OUT PROTOCOL INSTANCE

First, some wrappers

```
// Look for GOP implementations
auto locate_handle_buffer =
  wrap<3, 1, 1>(bootServices->LocateHandleBuffer);
// Create a GOP Instance
auto handle_protocol =
  wrap<2, 0, 1>(bootServices->HandleProtocol);
```

YAY, MONADS!

```
uint64_t handle_count = 0;
auto maybe_gop =
locate_handle_buffer(ByProtocol, graphicsOutProtocolGUID,
                   nullptr, handle_count)
 .and_then([&](auto proto_impl) {
     return handle_protocol(get<0>(proto_impl)[0],
                            graphicsOutProtocolGUID);
 })
 .map([](auto opened_gop) {
     return reinterpret_cast<EFI_GRAPHICS_OUTPUT_PROTOCOL *>(
      get<0>(opened_gop));
 });
if(!maybe_gop)
  conOut->OutputString(conOut, (CHAR16*) u"Fail\r\n");
```

FACT: This slide contains error handling

CREATE THE FRAMEBUFFER

```
gop = maybe_gop.value();
auto query_mode = wrap<2, 0, 2>(gop->QueryMode);
for(int i = 0;; ++i) {
  auto mode_info = query_mode(gop, i)
  .map([](auto r)->decltype(auto){return *std::get<1>(r);});
  if(!mode_info) break;
  if(mode_info->HorizontalResolution == width &&
     mode_info->VerticalResolution == height &&
     mode_info->PixelFormat ==
     PixelBlueGreenRedReserved8BitPerColor) {
       gop->SetMode(gop, i);
       break;
  }
}
```

USE THE FRAMEBUFFER

With emulated double buffering

```
void swap_to_screen() const {
  auto const pixel_ptr =
    reinterpret_cast<uint32_t *>(gop->Mode->FrameBufferBase);
  for(auto const& row : rows) {
    std::copy(row.begin(), row.end(), pixel_ptr);
    pixel_ptr += gop->Mode->Info->PixelsPerScanLine;
  }
}
void clear() {
  for(auto &row : rows) {
    std::fill(row.begin(), row.end(), T{});
  }
}
```

IMPLICIT SURFACES

Remember: Avoided heap

Implicit surfaces using signed distance fields offer a functional representation of a scene

Unfortunately no time for details

```
auto scene = [](vec3 const& rayT) {
  return pUnion(sphere(vec3( 10.0, 0.0, 50), 20, rayT),
                sphere(vec3(-10.0, 5.0, 50), 20, rayT));
};
```

CORRECT CODE?

We are in a freestanding environment, yet we use features that are not available there.

See Ben Craigs P0829 “Freestanding Proposal” for a solution.

A FUTURE WITH ZERO COST EXCEPTIONS

HERB SUTTER: P0709

“Zero-overhead deterministic exceptions”

```
constexpr auto make_out_param_adapter() {
  return [](args...) -> std::tuple<outs...> throws(EFI_STATUS) {
    // ...
    if(result != 0) {
        throw result;
    }
    // ...
  }
}
```

REFERENCES

- github.com/mmha/efiraytracer
- github.com/TartanLlama/expected
- OSDev Wiki: UEFI Bare Bones
- P0829r2: Freestanding Proposal