Is This Available?

Miro Knejp

Meeting C++ 2017

Platform-provided services

```
#if _WIN32
#if WINVER > 0xF00 && _ALIGNED_PLANETS < 5
class service { ... };
#elseif WINVER == MAGIC_VERSION
class service { ... };
#endif

#elseif __APPLE__ && MAGIC_MOUSE_CONNECTED
class service { ... };

#elseif _MATRIX
notify("Agent Smith");
class service { ... };

#elseif ...
...
```

What if not mutually exclusive

```
#if _WIN32
class win32_service { ... };

#if WINVER > 0xCAFE
class win32_service2 { ... };
#endif

#endif

#if MYLIB_USE_PYTHON_SERVICE
class python_service { ... };
#endif
```

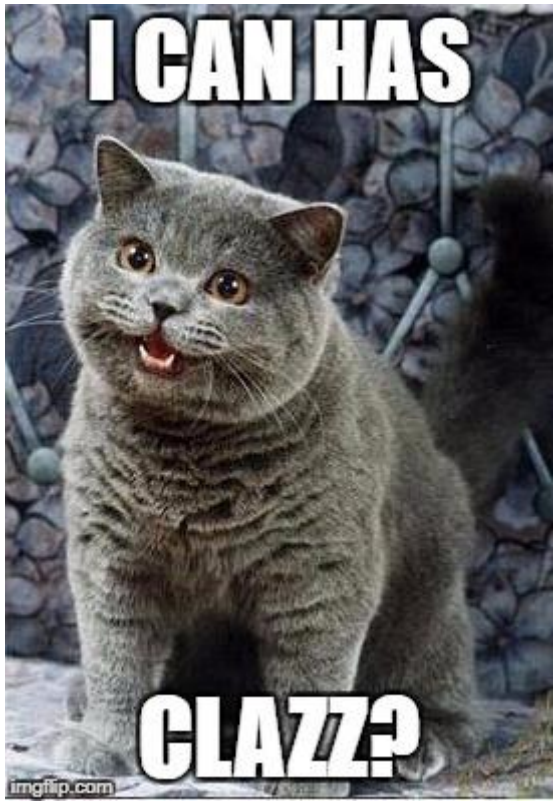
What if not mutually exclusive

```
#if _WIN32
class win32_service { ... };

#if WINVER > 0xCAFE
class win32_service2 { ... };
#endif

#endif

#if MYLIB_USE_PYTHON_SERVICE
class python_service { ... };
#endif
```



¯_(ツ)_/¯

Obvious solution

```
#if _WIN32
class win32_service { ... };
#define MYLIB_WIN32_SERVICE_AVAILABLE 1

#if WINVER > 0xCAFE
class win32_service2 { ... };
#define MYLIB_WIN32_SERVICE2_AVAILABLE 1
#endif

#endif

#if MYLIB_USE_PYTHON_SERVICE
class python_service { ... };
#define MYLIB_PYTHON_SERVICE_AVAILABLE 1
#endif
```

Obvious solution won’t work with Modules TS

```
#if _WIN32
class win32_service { ... };
#define MYLIB_WIN32_SERVICE_AVAILABLE 1

#if WINVER > 0xCAFE
class win32_service2 { ... };
#define MYLIB_WIN32_SERVICE2_AVAILABLE 1
#endif

#endif

#if MYLIB_USE_PYTHON_SERVICE
class python_service { ... };
#define MYLIB_PYTHON_SERVICE_AVAILABLE 1
#endif
```

```
import mylib;

// always fails
#if MYLIB_WIN32_SERVICE_AVAILABLE
...
#endif
```

```
struct available_service
{
  static constexpr std::true_type available;
};

struct unavailable_service
{
  static constexpr std::false_type available;
};
```

Announce your availability

```
struct available_service
{
  static constexpr std::true_type available;
};

struct unavailable_service
{
  static constexpr std::false_type available;
};


#if _WIN32
class win32_service : public available_service { ... };
#else
class win32_service : public unavailable_service {}; // has a definition
#endif
```

Announce your availability

```
struct available_service
{
  static constexpr std::true_type available;
};

struct unavailable_service
{
  static constexpr std::false_type available;
  void foo() { assert(false && "what are you doing?"); } // satisfy concept
};

#if _WIN32
class win32_service : public available_service { ... };
#else
class win32_service : public unavailable_service {}; // has a definition
#endif
```

Compile-time toolbox

```
using default_service =
  std::conditional_t<
    win32_service::available,
    win32_service,
    posix_service>;

if constexpr(win32_service::available && !python_service::available) { ... }
else { ... }

void use_service(python_service) requires python_service::available;
```

Pattern Matching!

Pattern Matching!

```
template<bool B> using has_a = std::bool_constant<B>;
template<bool B> using has_b = std::bool_constant<B>;
template<bool B> using has_c = std::bool_constant<B>;
```

Pattern Matching!

```
template<bool B> using has_a = std::bool_constant<B>;
template<bool B> using has_b = std::bool_constant<B>;
template<bool B> using has_c = std::bool_constant<B>;

auto select_service(has_a<true>,  has_b<false>, has_c<false>) -> service_a;
auto select_service(has_a<false>, has_b<true>,  has_c<false>) -> service_b;
```

Pattern Matching!

```
template<bool B> using has_a = std::bool_constant<B>;
template<bool B> using has_b = std::bool_constant<B>;
template<bool B> using has_c = std::bool_constant<B>;

auto select_service(has_a<true>,  has_b<false>, has_c<false>) -> service_a;
auto select_service(has_a<false>, has_b<true>,  has_c<false>) -> service_b;
...


using service = decltype(select_service(service_a::available,
                                        service_b::available,
                                        service_c::available));
```

Pattern Matching! With Wildcards!

```
template<bool B> using has_a = std::bool_constant<B>;
template<bool B> using has_b = std::bool_constant<B>;
template<bool B> using has_c = std::bool_constant<B>;

auto select_service(has_a<true>,  has_b<false>, has_c<false>) -> service_a;
auto select_service(dont_care,    has_b<true>,  has_c<false>) -> service_b;
auto select_service(dont_care,    has_b<true>,  has_c<true>)  -> void = delete;
auto select_service(dont_care,    dont_care,    has_c<true>)  -> service_c;

using service = decltype(select_service(service_a::available,
                                        service_b::available,
                                        service_c::available));
```

Pattern Matching! With Wildcards!

```
template<bool B> using has_a = std::bool_constant<B>;
template<bool B> using has_b = std::bool_constant<B>;
template<bool B> using has_c = std::bool_constant<B>;

auto select_service(has_a<true>,  has_b<false>, has_c<false>) -> service_a;
auto select_service(dont_care,    has_b<true>,  has_c<false>) -> service_b;
auto select_service(dont_care,    has_b<true>,  has_c<true>)  -> void = delete;
auto select_service(dont_care,    dont_care,    has_c<true>)  -> service_c;

using service = decltype(select_service(service_a::available,
                                        service_b::available,
                                        service_c::available));

struct dont_care
{
  template<typename T>
  dont_care(T&&) { }
};
```

Don’t force the preprocessor on your users.

The rest follows naturally.

@mknejp 

 https://cpplang.now.sh