The Makefile is provided with the module generated from getTemplateForDirectory. In the case of C++ server implementation, you need to configure the compilation and installation of a C++ library (Shared Object) file:

Makefile for C++

```
LIBRARIES:=<LibraryName>
<LibraryName>_OBJECTS = <ListOfObjectsToInclude> #.c and/or .cpp files
<LibraryName>_LIBS = <ListOfLibraryDependencies> #<Interface>Stubs acscomponent ServiceErr <Interface>Err #etc.
```

The first thing needed to implement a server, is to add the required include files:

C++ Imports

```
#ifndef _<FILENAME>_H
#define _<FILENAME>_H

#ifndef __cplusplus
#error This is a C++ include file and cannot be used from plain C
#endif

//Base component implementation, including container services and component lifecycle infrastructure
#include <acscomponentImpl.h>


//Skeleton interface for server implementation
#include <<Interface>S.h>


//Error definitions for catching and raising exceptions
#include <SYSTEMErr.h>
#include <<Interface>Err.h>

...


#endif
```

Besides this, there's the need to create the class using the mentioned infrastructure provided by the imported class and interface:

Server Class Definition

```
//ClassName usually is <Interface> or <Interface>Impl, but can be anything
class <ClassName>: public virtual acscomponent::ACSComponentImpl, public POA_<Module>::<Interface>
{
  public:
    <ClassName>(const ACE_CString& name, maci::ContainerServices * containerServices);
    ...
};
```

To access a struct, enum, typedef or definition in a module or in an interface, simply do the following:

C++ Types Usage

```
//From IDL <Module>::<EnumName>::<VALUE>
<Module>::<VALUE>;

#From IDL <Module>::<Interface>::<Enumname>::<VALUE>
<Module>::<Interface>::<VALUE>;
```

To retrieve a component to interact with, simply do the following:

C++ Component Interaction

```
//Shared
#include <IDLName>C.h;

//By Name
<Module>::<Interface>_var comp = this->getContainerServices()->getComponent<<Module>::<Interface>>("<Name>");

//By Interface. Must be at least one component configured as default!
<Module>::<Interface>_var comp = this->getContainerServices()->getDefaultComponent<<Module>::<Interface>>("IDL:
alma/<Module>/<Interface>:1.0");

//Release Components
this->getContainerServices()->releaseComponent(comp->name());
```

For logging in C++, there are some macros to help:

C++ Logger

```
ACS_TRACE("...");
ACS_DEBUG("...");
ACS_SHORT_LOG((LM_INFO, "..."));
ACS_SHORT_LOG((LM_WARNING, "..."));
ACS_SHORT_LOG((LM_ERROR, "..."));
```

For catching and raising exceptions:

C++ Error Handling

```
//Shared
#include <<Interface>Err.h>

//For catching exceptions
//Along CORBA calls
catch(<Interface>Err::<ExceptionName>Ex &_ex) { ... }
//Internally in the server... more convenient to edit parameters or log if needed
catch(<Interface>Err::<ExceptionName>ExImpl &_ex) { ... }

//For raising exceptions
//Along CORBA calls
throw <Interface>Err::<ExceptionName>ExImpl(__FILE__, __LINE__, "<CustomMessage>").get<ExceptionName>Ex();
//Internally in the server...
throw <Interface>Err::<ExceptionNanem>ExImpl(__FILE__, __LINE__, "<CustomMessage>");

//For raising exceptions with parameters
<Interface>Err::<ExceptionName>ExImpl err(__FILE__, __LINE__, "<CustomMessage>");
err.set<ParamName>(<Value>);
throw err.get<ExceptionName>Ex();

//For logging an error message from the exceptions
<Interface>Err::<ExceptionName>ExImpl err(__FILE__, __LINE__, "<CustomMessage>");
err.log();
throw err.get<ExceptionName>Ex();
```

Data types mapping:

To check how data types are mapped between IDL and implementation click here.