OPCPP

GETTING STARTED GUIDE

GETTING STARTED

What is opCpp?

opCpp is a general purpose translating compiler. It takes an extended form of c++ as input and outputs the original code and set of interspersed annotations which can be turned into concrete c++ code by the user. This is a safe process which adds no runtime overhead to c++ code, and can be customized by the user. opCpp can improve productivity by cutting the amount of glue and redundant code required by manual code annotations.

Core Features:

- 1. code annotation generation (metadata for standard c++)
- 2. compile-time, no overhead (no libraries)
- 3. customizable syntax and validation
- 4. intuitive tool & IDE integration

What problems can opCpp help solve?

The main function of opCpp is to provide additional data to the backend c++ compiler. In addition to providing additional data it adds a form of attribute support to c++. This makes opCpp well suited to any number of problems handled in dynamic languages by attribute-oriented programming, including...

Serialization

The main reason to use opCpp is for implementing serialization mappings on new or existing c++ projects. By using opCpp, you can automate your serialization mappings, and still get syntax validation.

Reflection

You may also use opCpp to generate mappings for a data reflection system. The amount of data, types, or functions you expose is entirely your decision.

Bindings

opCpp may be used to generate automatic bindings to non-native languages or scripting languages.

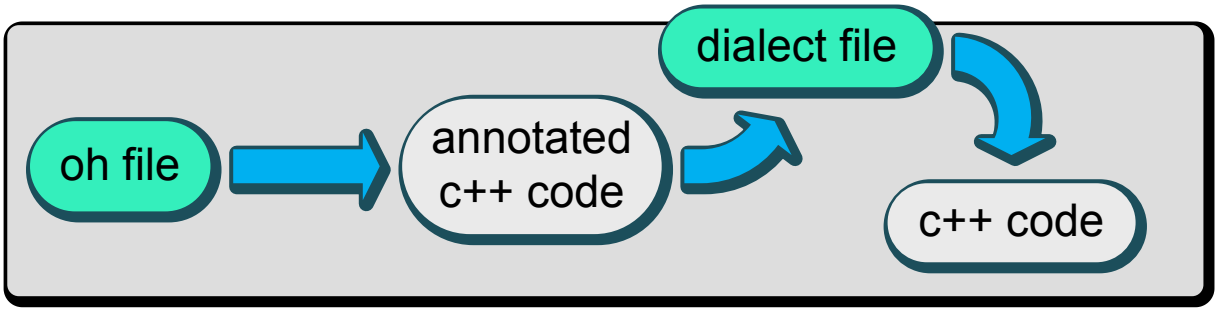
C++ Concerns

In addition, the compile-time nature of c++ requires us to output extra annotations and keep in mind portability for backend compilers. opCpp helps handle concerns related to source and header files, inflexible c++ syntax, and other issues typically handled with cryptic macros.

THE OPCPP PROCESS

Compilation Stages

When using opCpp, you deal directly with *oh files* – these are the code files you'll write. In effect oh files are just improved c++ header files. These are fed into opCpp which annotates the code, and converts any extended c++ syntax into valid c++ syntax. The second thing you'll write are *dialect files*. Dialects connect the annotations to the actual code generated, and you may think of them as a simple substitution transformation. In opCpp Version 1, dialects are in the form of preprocessor definitions – this may change in the future to allow more advanced control.



Tool Support

opCpp generated code is debuggable by all major debugging faculties – in fact it's more debuggable than c++. The reason for this is anytime you provide additional keywords, they may be tied to additional underlying code. Because of this interesting fact, if you have code tied to serialization of a data member you can in fact breakpoint on the data members! This has fascinating implications, and we think this is one of the most interesting things about opCpp.

Because opCpp code is translated, backend compiler errors are also a potential issue. We address this, and errors related to the generated code will resolve to the original oh file, which is generally what you want.

We have also provided built in update detection and indexing of opCpp generated output. This means you can use opCpp to handle your entire oh file related build process, with a minimal amount of work to integrate your generated code. We are creating add-ins for Visual Studio 2005 as well, and we expect using opCpp to be transparent once the add-ins are complete.

THE OPCPP PACKAGE

The Full Package Includes

- opCpp compiler
- opCpp headers and default dialects
- *The opCpp Getting Started Guide* (this document)
- *The opCpp Manual*
- Visual Studio 2005 Integration
- Example projects

Example Projects

The remainder of this document is related to the example projects. These are found in the opCpp install directory relative path /opcpp/examples/. The examples are explained starting in each project .cpp file. Read this document while reading/stepping through the source to help understand the examples.

The examples diagrammed in this document are:

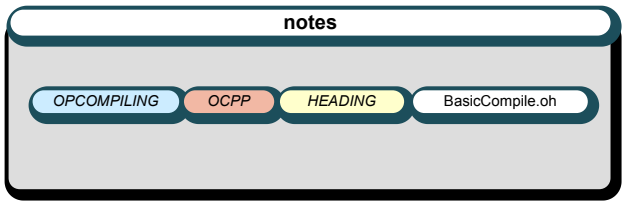
1. Basic Compile:
shows printing mappings and compile ordering.

2. Dialect Compile:
shows how to generate a dialect through opCpp and print xml.

3. File Serialize:
shows how to setup a serialization scheme in opCpp, and one way of setting up saving, loading and printing (with textual enumerations).

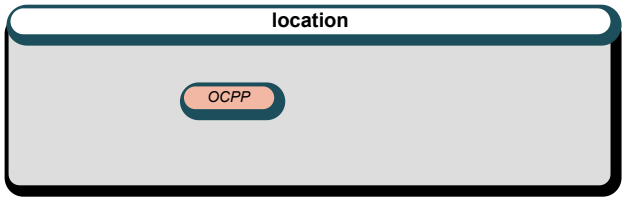
LEGEND FOR NOTE DIAGRAMS

opCpp automatically generates *notes*. They allow you to substitute code in a particular code region. In the diagrams we color each part of the note to show its meaning.



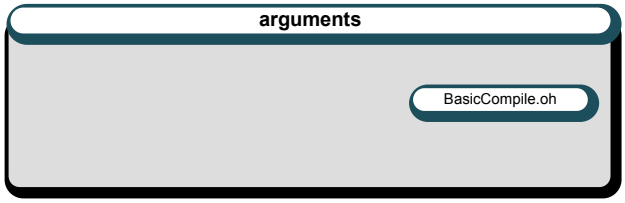
The first part of an opCpp note is the *feature*. This particular note is related to compiling, while other features are type-related (including opclass, opstruct, openum).

The second part of a note is the *location*. Because C++ is often order dependant, it's important to identify where our notes are located.

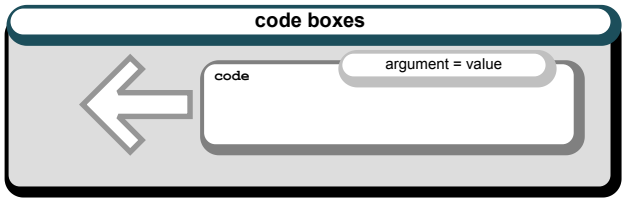


The third part of a note is a *tag*. There may be more than one tag per note. They're usually used to distinguish a note, but may provide extra information, which would be less useful in argument form.

Some notes have *arguments*. In the diagrams we show the arguments opCpp generates for the examples.

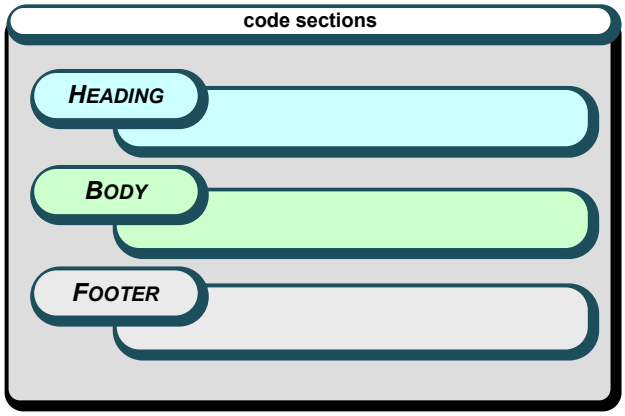


These arrowed boxes show the code that is substituted for a certain note. If arguments are used, this shows the arguments values also. Notes and code are tied together via opCpp *dialects*. Some arrows are removed for clarity.

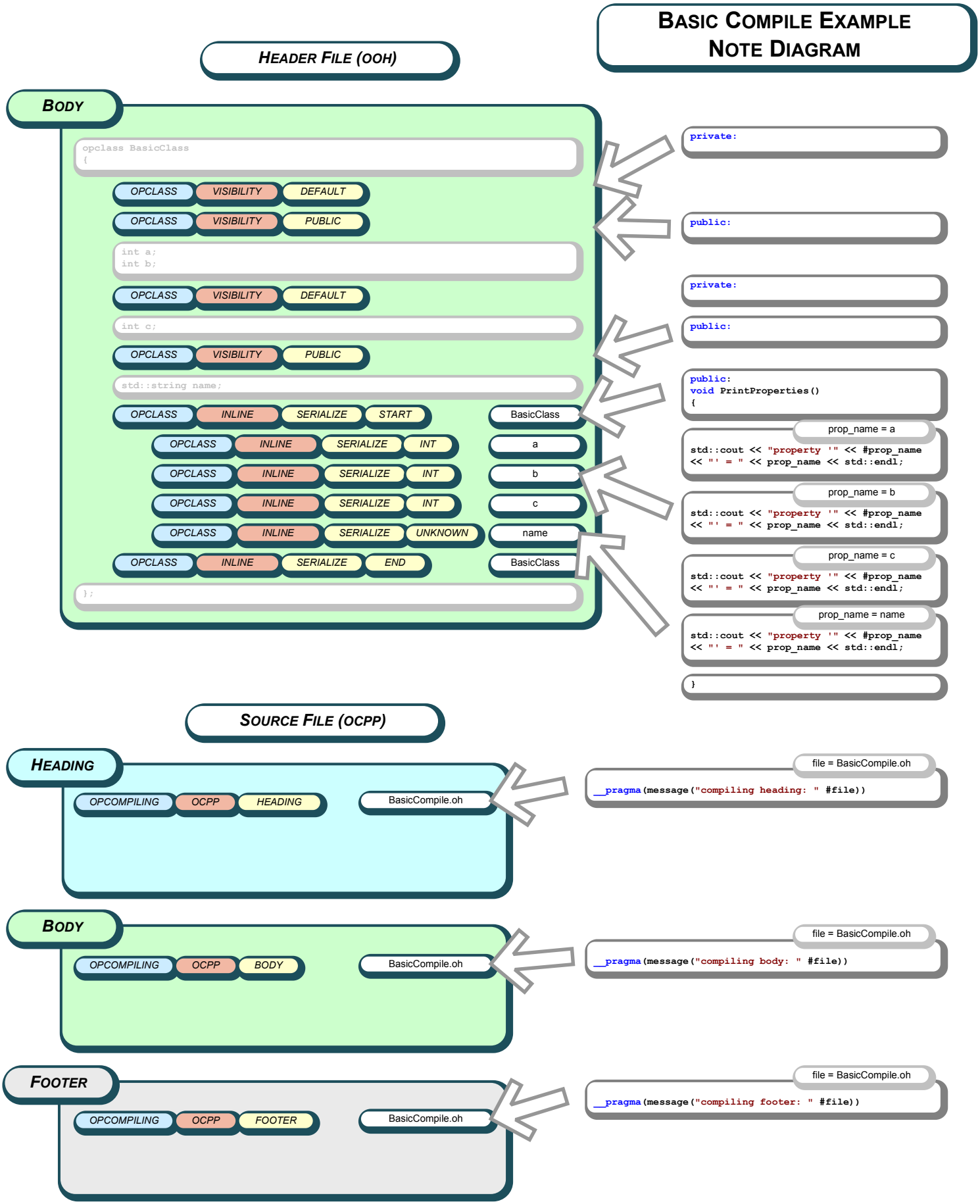


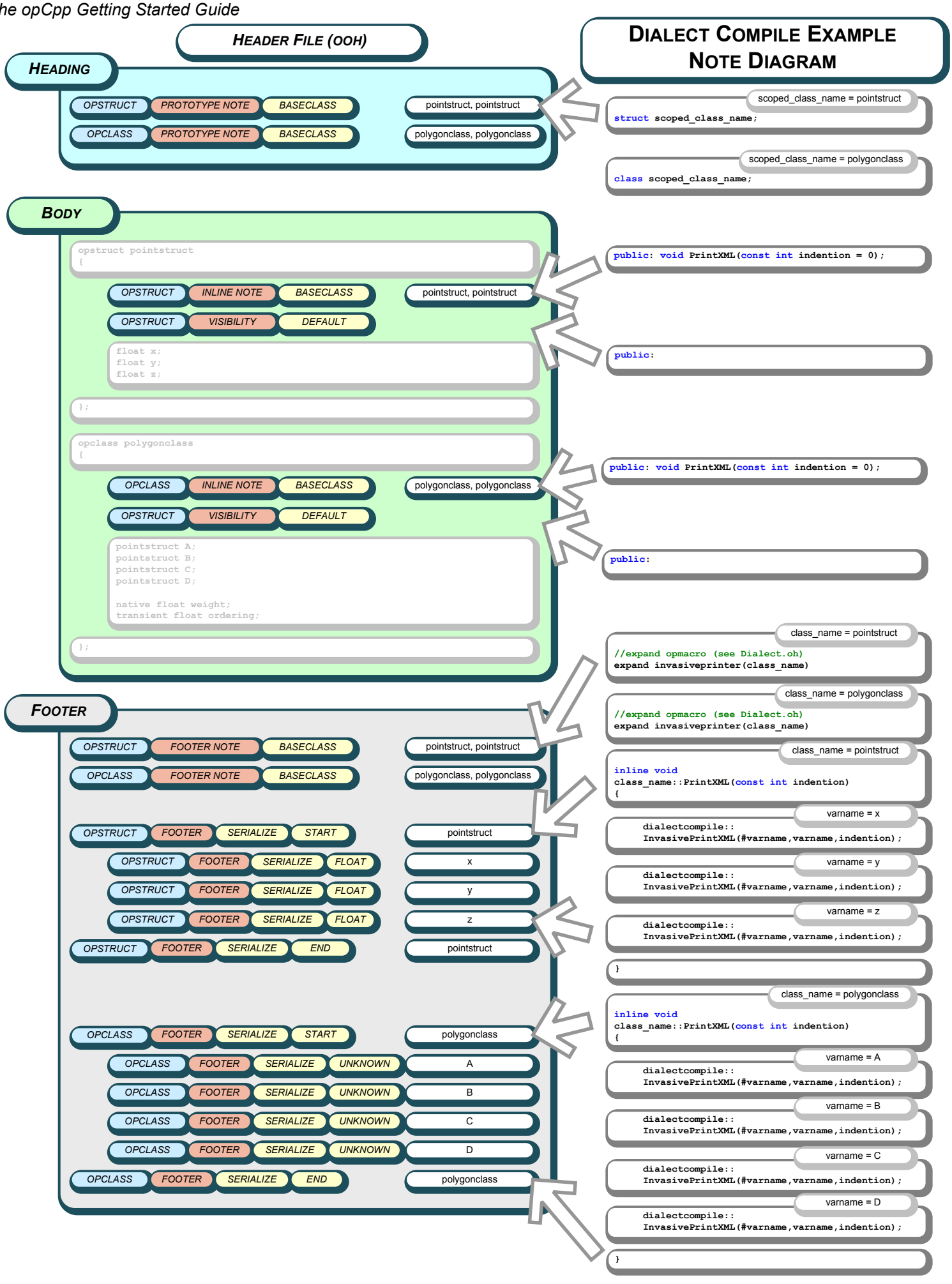
These colored boxes show the sections in the opCpp transformed output. Particular notes go to either the output header (ooh) or output source (ocpp) files, these show their placement within those files.

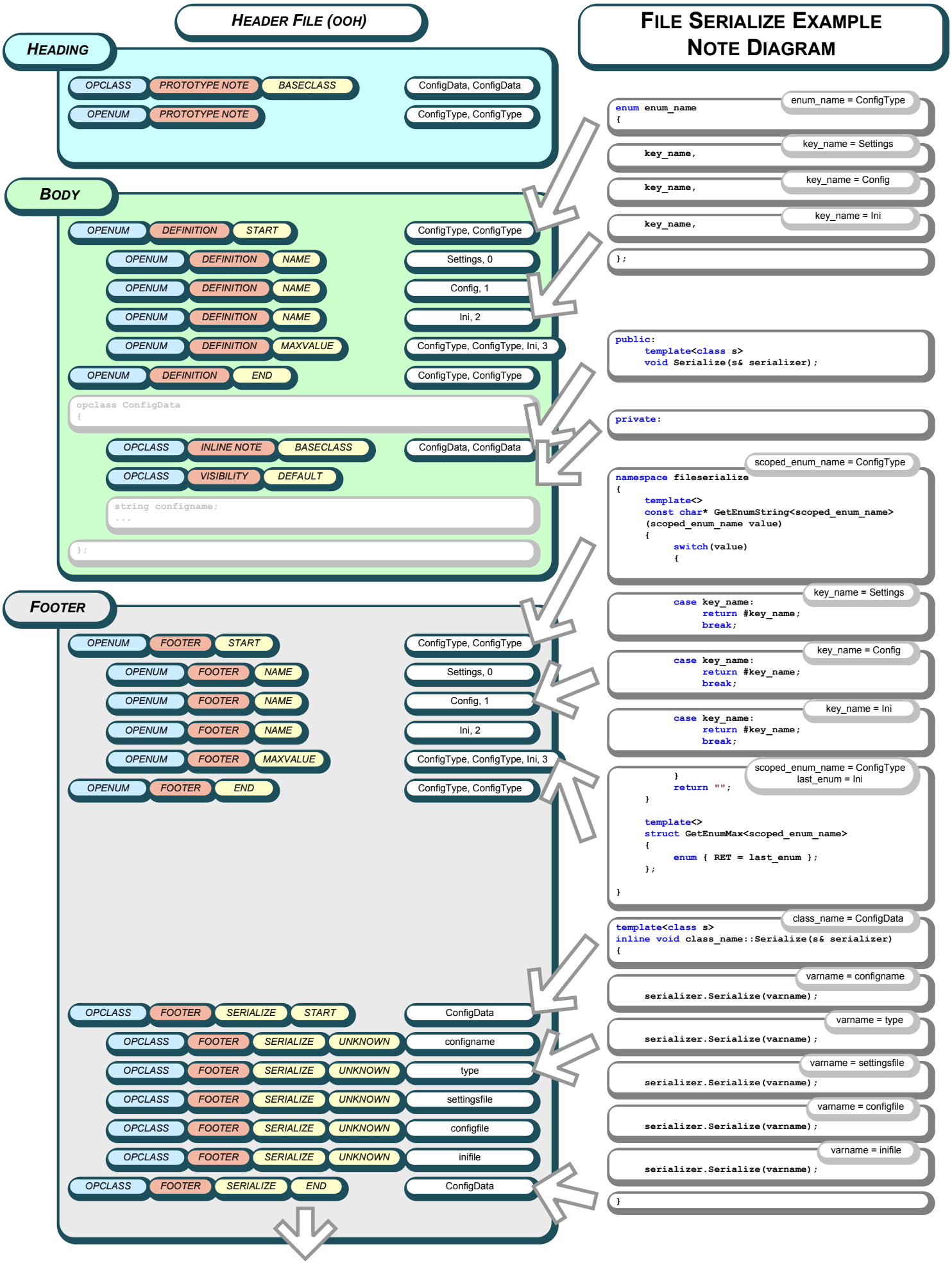
Also, these three sections are compiled separately across all generated files, meaning all headings are compiled before any body code is seen by the c++ compiler.

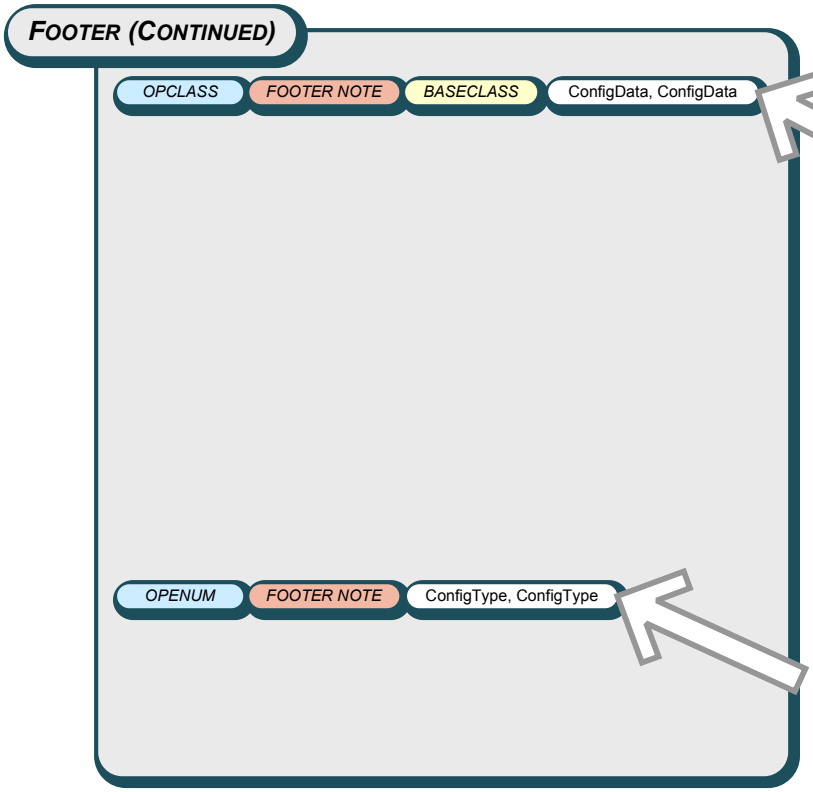


Basic Compile Example Note Diagram










```
namespace fileserialize
{
     template<>
     inline void LoadSerializer::Serialize<scoped_class_name>
     (scoped_class_name& element)
     {
          element.Serialize(*this);
     }

     template<>
     inline void SaveSerializer::Serialize<scoped_class_name>
     (scoped_class_name& element)
     {
          element.Serialize(*this);
     }

     template<>
     inline void PrintSerializer::Serialize<scoped_class_name>
     (scoped_class_name& element)
     {
          element.Serialize(*this);
     }
}
```

```
namespace fileserialize
{
     template<>
     inline void LoadSerializer::Serialize<scoped_enum_name>
     (scoped_enum_name& element)
     {
          //compressed version loading
          //zero the element since we may
          //only read in some of the bytes
          element = (scoped_enum_name)0;
          //this will write 1 byte usually
          filestream.read(reinterpret_cast<char*>(&element),
               fileserialize::GetMinEnumBytes<scoped_enum_name>::RET
               );
          }

     template<>
     inline void SaveSerializer::Serialize<scoped_enum_name>
     (scoped_enum_name& element)
     {
          //compressed version saving
          //this will write 1 byte usually
          filestream.write(reinterpret_cast<const char*>(&element),
               fileserialize::GetMinEnumBytes<scoped_enum_name>::RET
               );
     }
}
```