The gSOAP Stub and Skeleton Compiler for C and C++ 2.1.6

Robert A. van Engelen
Department of Computer Science
Florida State University
Tallahassee, FL32306-4530
engelen@cs.fsu.edu

July 14, 2002

Contents

1 Introduction

The gSOAP toolkit provides a unique SOAP-to-C/C++ language binding for the development of SOAP Web Services and clients. Other SOAP C++ implementations adopt a SOAP-centric view and offer SOAP APIs for C++ that require the use of class libraries for SOAP-like data structures. This often forces a user to adapt the application logic to these libraries. In contrast, gSOAP provides a C/C++ transparent SOAP API through the use of compiler technology that hides irrelevant SOAP-specific details from the user. The gSOAP stub and skeleton compiler automatically maps native and user-defined C and C++ data types to semantically equivalent SOAP data types and vice-versa. As a result, full SOAP interoperability is achieved with a simple API relieving the user from the burden of SOAP details and enables him or her to concentrate on the application-essential logic. The compiler enables the integratation of (legacy) C/C++ and Fortran codes (through a Fortran-to-C interface), embedded systems, and real-time software in SOAP applications that share computational resources and information with other SOAP applications, possibly across different platforms, language environments, and disparate organizations located behind firewalls.

gSOAP minimizes application adaptation for building SOAP clients and Web Services. The gSOAP compiler generates SOAP marshalling routines that (de)serialize application-specific C/C++ data structures. gSOAP includes a WSDL generator to generate Web service descriptions for your Web services. The gSOAP WSDL importer "closes the circle" in that it enables client development without the need for users to analyze Web service details to implement a client.

Some of the highlights of gSOAP are:

- Unique interoperability features: the gSOAP compiler generates SOAP marshalling routines that (de)serialize native and user-defined C/C++ data structures. gSOAP is also one of the few SOAP toolkits that support the full range of SOAP 1.1 features including multi-dimensional arrays and polymorphic types. For example, a remote method with a base class parameter may accept derived class instances from a client. Derived class instances keep their identity through dynamic binding.
- gSOAP includes a WSDL generator for convenient Web Service publishing.
- gSOAP includes a WSDL importer for automated client development.
- Generates source code for stand-alone Web Services and clients.
- Ideal for building web services that are compute-intensive and are therefore best written in C and C++.
- Platform independent: Windows, Unix, Linux, Pocket PC, etc.
- Fast *in situ* serialization and deserialization with SOAP encoding of arbitrary user-defined and built-in C and C++ data structures.
- Fully SOAP 1.1 compliant data encoding and decoding. (Also SOAP 1.2 compliant, except for header faults, SOAP actors, SOAP root.)
- DIME compliant attachments.
- The schema-specific XML pull parser is fast and efficient and does not require intermediate data storage for demarshalling to save space and time.

- Selective input and output buffering is used to increase efficiency, but full message buffering to determine HTTP message length is not used. Instead, a three-phase serialization method is used to determine message length. As a result, large data sets such as base64-encoded images can be transmitted with or without DIME attachments by small-memory devices such as PDAs.

- Supports C++ single class inheritance, dynamic binding, overloading, arbitrary pointer structures such as lists, trees, graphs, cyclic graphs, fixed-size arrays, (multi-dimensional) dynamic arrays, enumerations, built-in XML schema types including base64Binary encoding, and hexBinary encoding.

- No need to rewrite existing C/C++ applications for Web service deployment. However, parts of an application that use unions, pointers to sequences of elements in memory, and **void*** need to be modified, but **only** if the data structures that adopt them are required to be serialized or deserialized as part of a remote method invocation.

- Three-phase marshalling: 1) analysis of pointers, single-reference, multi-reference, and cyclic data structures, 2) HTTP message-length determination, and 3) serialization as per SOAP 1.1 encoding style or user-defined encoding styles.

- Two-phase demarshalling: 1) SOAP parsing and decoding, which involves the reconstruction of multi-reference and cyclic data structures from the payload, and 2) resolution of "forward" pointers (i.e. resolution of the forward `href` attributes in SOAP).

- Full and customizable SOAP Fault processing (client receive and service send).

- Customizable SOAP Header processing (send and receive), which for example enables easy transaction processing for the service to keep state information.

2 Notational Conventions

The typographical conventions used by this document are:

Sans serif or italics font Denotes C and C++ source code, file names, and commands.

Bold font Denotes C and C++ keywords.

`Courier font` Denotes HTTP header content, HTML, XML, and XML schema fragments.

[Optional] Denotes an optional construct.

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119.

3 Differences Between gSOAP Versions 1.X and 2.X

gSOAP versions 2.0 and higher are redesigned and thread-safe. All files in the gSOAP 2.X distribution are renamed to avoid confusion with gSOAP version 1.X files:

gSOAP 1.X	**gSOAP 2.X**
soapcpp	soapcpp2
soapcpp.exe	soapcpp2.exe
stdsoap.h	stdsoap2.h
stdsoap.c	stdsoap2.c
stdsoap.cpp	stdsoap2.cpp

Changing the version 1.X application codes to accomodate gSOAP 2.X does not require a significant amount of recoding. The change to gSOAP 2.X affects all functions defined in stdsoap2.c[pp] (the gSOAP runtime environment API) and the functions in the sources generated by the gSOAP compiler (the gSOAP RPC+marshalling API). Therefore, clients and services developed with gSOAP 1.X need to be modified to accomodate a change in the calling convention used in 2.X: In 2.X, **all** gSOAP functions (including the remote method proxy routines) take an additional parameter which is an instance of the gSOAP runtime environment that includes file descriptors, tables, buffers, and flags. This additional parameter is **always** the first parameter of any gSOAP function.

The gSOAP runtime environment is stored in a **struct** soap type. A **struct** was chosen to support application development in C without the need for a separate gSOAP implementation. An object-oriented approach with a class for the gSOAP runtime environment would have prohibited the implementation of pure C applications. Before a client can invoke remote methods or before a service can accept requests, a runtime environment need to be allocated and initialized. Two new functions are added to gSOAP 2.X:

Function	**Description**
soap_init(**struct** soap *soap)	Initializes a runtime environment (required only once)
struct soap *soap_new()	Allocates, initializes, and returns a pointer to a runtime environment

An environment can be reused as many times as necessary and does not need to be reinitialized in doing so. A new environment is only required for each new thread to guarantee exclusive access to a new runtime environment by each thread. For example, the following code stack-allocates the runtime environment which is used for multiple remote method calls:

```
int main()
{
  struct soap soap;
  ...
  soap_init(&soap); // initialize runtime environment
  ...
  soap_call_ns__method1(&soap, ...); // make a remote call
  ...
  soap_call_ns__method2(&soap, ...); // make another remote call
  ...
  soap_end(&soap); // clean up
  ...
}
```

The runtime environment can also be heap allocated:

```
int main()
{
  struct soap *soap;
  ...
  soap = soap_new(); // allocate and initialize runtime environment
  if (!soap) // couldn't allocate: stop
  ...
  soap_call_ns__method1(soap, ...); // make a remote call
  ...
  soap_call_ns__method2(soap, ...); // make another remote call
  ...
  soap_end(soap); // clean up
  ...
  free(soap); // deallocate runtime environment
}
```

A service need to allocate and initialize an environment before calling soap_serve:

```
int main()
{
  struct soap soap;
  soap_init(&soap);
  soap_serve(&soap);
}
```

Or alternatively:

```
int main()
{
  soap_serve(soap_new());
}
```

A service can use multi-threading to handle requests while running some other code that invokes remote methods:

```
int main()
{
  struct soap soap1, soap2;
  pthread_t tid;
  ...
  soap_init(&soap1);
  if (soap_bind(&soap1, host, port, backlog) < 0) exit(-1);
  if (soap_accept(&soap1) < 0) exit(-1);
  pthread_create(&tid, NULL, (void*(*)(void*))soap_serve, (void*)&soap1]);
  ...
  soap_init(&soap2);
  soap_call_ns__method(&soap2, ...); // make a remote call
  ...
  soap_end(&soap2);
  ...
  pthread_join(tid); // wait for thread to terminate
  soap_end(&soap1); // release its data
}
```

In the example above, two runtime environments are required. In comparison, gSOAP 1.X statically allocates the runtime environment, which prohibits multi-threading (only one thread can invoke remote methods and/or accept requests due to the single runtime environment).

Section 5.2.3 presents a multi-threaded stand-alone Web Service that handles multiple SOAP requests by spawning a thread for each request.

4 Interoperability

gSOAP interoperability has been verified with the following SOAP implementations and toolkits:

Apache 2.2

Apache Axis

ASP.NET

Cape Connect

Delphi

easySOAP++

eSOAP

Frontier

GLUE

Iona XMLBus

kSOAP

MS SOAP

Phalanx

SIM

SOAP::Lite

SOAP4R

Spray

SQLData

Wasp Adv.

Wasp C++

White Mesa

xSOAP

ZSI

4S4C

5 Quick User Guide

This user guide offers a quick way to get started with gSOAP. This section requires a basic understanding of the SOAP 1.1 protocol and some familiarity with C and/or C++. In principle, SOAP clients and SOAP Web services can be developed in C and C++ with the gSOAP stub and skeleton compiler without a detailed understanding of the SOAP protocol when the applications are build as an ensamble and only communicate within this group (i.e. meaning that you don't have to worry about interoperability with other SOAP implementations). This section is intended to illustrate the implementation of SOAP C/C++ Web services and clients that connect to other existing SOAP implementations such as Apache SOAP and SOAP::Lite for which some details of the SOAP protocol need to be understood.

5.1 How to Use the gSOAP Stub and Skeleton Compiler to Build SOAP Clients

In general, the implementation of a SOAP client application requires a **stub** routine for each remote method that the client application needs to invoke. The primary stub's responsibility is to marshall the input data, send the request to the designated SOAP service over the wire, to wait for the response, and to demarshall the output data when it arrives. The client application invokes the stub routine for a remote method as if it would invoke a local method. To write a stub routine in C or C++ by hand is a tedious task, especially if the input and/or output data structures of a remote method are complex data types such as records, arrays, and graphs.

The generation of stub routines for a SOAP client is fully automated with gSOAP. The gSOAP stub and skeleton compiler is a **preprocessor** that generates the necessary C++ sources to build SOAP C++ clients. The input to the gSOAP stub and skeleton compiler consists of a standard C/C++ **header file**. The header file can be generated from a WSDL (Web Service Description Language) documentation of a service with the gSOAP WSDL importer, see 5.2.7. The SOAP remote methods are specified in this header file as **function prototypes**. Stub routines in C/C++ source form are automatically generated by the gSOAP compiler for these function prototypes of remote methods. The resulting stub routines allow C and C++ client applications to seamlessly interact with existing SOAP Web services.

The gSOAP stub and skeleton compiler also generates **skeleton** routines for each of the remote methods specified in the header file. The skeleton routines can be readily used to implement one or more of the remote methods in a new SOAP Web service. These skeleton routines are not used for building SOAP clients in C++, although they can be used to build mixed SOAP client/server applications (peer applications).

The input and output parameters of a SOAP remote method may be simple data types or complex data types. The necessary **type declarations** of C/C++ user-defined data structures such as structs, classes, enumerations, arrays, and pointer-based data structures (graphs) are to be provided in the header file. The gSOAP stub and skeleton compiler automatically generates **serializers** and **deserializers** for the data types to enable the generated stub routines to encode and decode the contents of the parameters of the remote methods.

The remote method name and its parameterization can be found with a SOAP Web service description, typically in the form of an XML schema. There is an almost one-to-one correspondence between the XML schema description of a SOAP remote method and the C++ type declarations required to build a client application for the Web service. The schemas are typically part of the

WSDL specification of a SOAP Web service. The gSOAP WSDL importer converts WSDL service descriptions into header files.

5.1.1 Example

The getQuote remote method of XMethods Delayed Stock Quote service provides a delayed stock quote for a given ticker name, see `http://xmethods.com/detail.html?id=2` for details. The WSDL description of the Delayed Stock Quote service provides the following details:

Endpoint URL:	`http://services.xmethods.net:80/soap`
SOAP action:	"" (2 quotes)
Remote method namespace:	`urn:xmethods-delayed-quotes`
Remote method name:	`getQuote`
Input parameter:	`symbol` of type `xsd:string`
Output parameter:	`Result` of type `xsd:float`

The following getQuote.h **header file** is created from the WSDL description with the WSDL importer:

```
// Content of file "getQuote.h":
int ns1__getQuote(char *symbol, float &Result);
```

The header file essentially specifies the remote method in C++. The remote method is declared as a ns1__getQuote **function prototype** which specifies all of the necessary details for the gSOAP compiler to generate the stub routine for a client application to interact with the Delayed Stock Quote service.

The Delayed Stock Quote service description requires that the **input parameter** of the getQuote remote method is a symbol parameter of type string. The description also indicates that the Result **output parameter** is a floating point number that represents the current unit price of the stock in dollars. The gSOAP compiler uses the convention the **last parameter** of the function prototype must be the output parameter of the remote method, which is required to be passed by reference using the reference operator (&) or by using a pointer type. All other parameters except the last are input parameters of the remote method, which are required to be passed by value or passed using a pointer to a value (by reference is not allowed). The function prototype associated with a remote method is required to return an **int**, whose value indicates to the caller whether the connection to a SOAP Web service was successful or resulted in an exception, see Section 7.2 for the error codes.

The use of the namespace prefix ns1__ in the remote method name in the function prototype declaration is discussed in detail in 5.1.2. Basically, a namespace prefix is distinghuished by a **pair of underscores** in the function name, as in ns1__getQuote where ns1 is the namespace prefix and getQuote is the remote method name. (A single underscore in an identifier name will be translated to a dash in the XML output, see Section 7.3.)

The gSOAP compiler is invoked from the command line with:

```
soapcpp2 getQuote.h
```

The compiler generates the stub routine for the getQuote remote method specified in the getQuote.h header file. This stub routine can be called by a client program at any time to request a stock quote from the Delayed Stock Quote service. The interface to the generated stub routine is called a **proxy**, which is the following function generated by the gSOAP compiler:

```
int soap_call_ns1__getQuote(struct soap *soap, char *URL, char *action, char *symbol, float
&Result);
```

The stub routine of the proxy is saved in soapClient.cpp. The file soapC.cpp contains the **serializer** and **deserializer** routines for the data types used by the stub.

Note that the parameters of the soap_call_ns1__getQuote proxy are identical to the ns1__getQuote function prototype with three additional input parameters: soap must be a valid pointer to a gSOAP runtime environment, URL is the SOAP Web service **endpoint URL** passed as a string, and action is a string that denotes the **SOAP action** required by the Web service.

The following example C++ client program invokes the proxy to retrieve the latest AOL stock quote from the XMethods Delayed Stock Quote service:

```
#include "soapH.h" // obtain the generated proxy
int main()
{
  struct soap soap; // gSOAP runtime environment
  float quote;
  soap_init(&soap); // initialize runtime environment (only once)
  if (soap_call_ns1__getQuote(&soap, "http://services.xmethods.net:80/soap", "", "AOL",
quote) == SOAP_OK)
    cout << "Current AOL Stock Quote = " << quote;
  else // an error occurred
    soap_print_fault(&soap, stderr); // display the SOAP fault message on the stderr stream
soap_end(&soap); // clean up
}
```

The XMethods Delayed Stock Quote service endpoint URL is `http://services.xmethods.net/soap` port 80 and the SOAP action required is `""` (two quotes). If successful, the proxy returns SOAP_OK and quote contains the latest stock quote. Otherwise, an error occurred and the SOAP fault is displayed with the soap_print_fault function.

When the example client application is invoked, the SOAP request is performed by the proxy routine soap_call_ns1__getQuote, which generates the following SOAP request message:

```
POST /soap HTTP/1.1
Host: services.xmethods.net
Content-Type: text/xml
Content-Length: 529
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
  xmlns:ns1="urn:xmethods-delayed-quotes"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<ns1:getQuote>
<symbol>AOL</symbol>
</ns1:getQuote>
```

```
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The XMethods Delayed Stock Quote service responds with the SOAP response message:

```
HTTP/1.1 200 OK
Date: Sat, 25 Aug 2001 19:28:59 GMT
Content-Type: text/xml
Server: Electric/1.0
Connection: Keep-Alive
Content-Length: 491

<?xml version='1.0' encoding='UTF-8'?>
<soap:Envelope xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:xsi='http://www.w3.org/1999/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/1999/XMLSchema'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
<soap:Body>
<n:getQuoteResponse xmlns:n='urn:xmethods-delayed-quotes'>
<Result xsi:type='xsd:float'>41.81</Result>
</n:getQuoteResponse>
</soap:Body>
</soap:Envelope>
```

The server's response is parsed by the stub routine of the proxy of the client. The stub routine further demarshalls the data of `Result` element of the SOAP response and stores it in the quote parameter of soap_call_ns1__getQuote.

A client program can invoke a remote method at any time and multiple times if necessary. Consider for example:

```
...
struct soap soap;
float quotes[3]; char *myportfolio[] = {"IBM", "AOL", "MSDN"};
soap_init(&soap); // need to initialize only once
for (int i = 0; i < 3; i++)
  if (soap_call_ns1__getQuote(&soap, "http://services.xmethods.net:80/soap", "", myport-
folio[i], quotes[i]) != SOAP_OK)
    break;
if (soap.error) // an error occurred
  soap_print_fault(&soap, stderr);
soap_end(&soap); // clean up all deserialized data
...
```

This client composes an array of stock quotes by calling the ns1__getQuote proxy for each symbol in a portfolio array.

This example demonstrated how easy it is to build a SOAP client with gSOAP once the details of a Web service are available in the form of a WSDL document.

5.1.2 Namespace Considerations

The declaration of the ns1__getQuote function prototype (discussed in the previous section) uses the namespace prefix ns1__ of the remote method namespace, which is distinghuished by a **pair of underscores** in the function name to separate the namespace prefix from the remote method name. The purpose of a namespace prefix is to associate a remote method name with a service in order to prevent naming conflicts, e.g. to distinguish identical remote method names used by different services.

Note that the XML response of the XMethods Delayed Stock Quote service example uses the **namespace prefix** n which is associated with the **namespace URI** urn:xmethods-delayed-quotes through the xmlns:n="urn:xmethods-delayed-quotes binding. The use of namespace prefixes and namespace URIs is also required to enable SOAP applications to validate the content of a client's request and vice versa. The namespace URI in the service response is verified by the stub routine by using the information supplied in a **namespace mapping table** that is required to be part of the client program. The table is accessed at run time to resolve namespace bindings, both by the generated stub's data structure serializer for encoding the client request and by the generated stub's data structure deserializer to decode and validate the service response. The namespace mapping table should not be part of the header file input to the gSOAP stub and skeleton compiler.

The namespace mapping table for the Delayed Stock Quote client is:

```
struct Namespace namespaces[] =
{   // {"ns-prefix", "ns-name"}
   {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"}, // MUST be first
   {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"}, // MUST be second
   {"xsi", "http://www.w3.org/2001/XMLSchema-instance"}, // MUST be third
   {"xsd", "http://www.w3.org/2001/XMLSchema"}, // 2001 XML schema
   {"ns1", "urn:xmethods-delayed-quotes"}, // given by the service description
   {NULL, NULL} // end of table
};
```

The first four namespace entries in the table consist of the standard namespaces used by the SOAP 1.1 protocol. In fact, the namespace mapping table is explicitly declared to enable a programmer to specify the SOAP encoding style and to allow the inclusion of namespace-prefix with namespace-name bindings to comply to the namespace requirements of a specific SOAP service. For example, the namespace prefix ns1, which is bound to urn:xmethods-delayed-quotes by the namespace mapping table shown above, is used by the generated stub routine to encode the getQuote request. This is performed automatically by the gSOAP compiler by using the ns1 prefix of the ns1__getQuote method name specified in the getQuote.h header file. In general, if a function name of a remote method, **struct** name, **class** name, **enum** name, or field name of a **struct** or **class** has a pair of underscores, the name has a namespace prefix that must be defined in the namespace mapping table.

The namespace mapping table will be output as part of the SOAP Envelope by the stub routine. For example:

```
...
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
```

```
    xmlns:xsd="http://www.w3.org/1999/XMLSchema"
    xmlns:ns1="urn:xmethods-delayed-quotes"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
...
```

The namespace bindings will be used by a SOAP service to validate the SOAP request.

5.1.3 Example

The incorporation of namespace prefixes into C++ identifier names is necessary to distinguish remote methods that share the same name but are implemented in different Web services. Consider for example:

```
// Contents of file "getQuote.h":
int ns1__getQuote(char *symbol, float &Result);
int ns2__getQuote(char *ticker, char *&quote);
```

The namespace prefix is separated from the remote method names by a pair of underscores (__) by convention.

This example enables a client program to connect to a (hypothetical) Stock Quote service with remote methods that can only be distinghuished by their namespaces. Consequently, two different namespace prefixes have been used as part of the remote method names.

The namespace prefix convention can also be applied to **class** declarations that contain SOAP compound values that share the same name but have different namespaces that refer to different XML schemas. For example:

```
class e__Address // an electronic address
{
  char *email;
  char *url;
};
class s__Address // a street address
{
  char *street;
  int number;
  char *city;
};
```

The namespace prefix is separated from the data type names by a pair of underscores (__) by convention.

An instance of e__Address is encoded by the generated serializer for this type as an Address element with namespace prefix e:

```
<e:Address xsi:type="e:Address">
<email xsi:type="string">me@home</email>
<url xsi:type="string">www.me.com</url>
</e:Address>
```

While an instance of s__Address is encoded by the generated serializer for this type as an Address element with namespace prefix s:

```
<s:Address xsi:type="s:Address">
<street xsi:type="string">Technology Drive</street>
<number xsi:type="int">5</number>
<city xsi:type="string">Softcity</city>
</s:Address>
```

The namespace mapping table of the client program must have entries for e and s that refer to the XML schemas of the data types:

```
struct Namespace namespaces[] =
{ ...
  {"e", "http://www.me.com/schemas/electronic-address"},
  {"s", "http://www.me.com/schemas/street-address"},
...
```

This table is required to be part of the client application to allow access by the serializers and deserializers of the data types at run time.

5.1.4 Some SOAP Encoding Considerations

Many SOAP services require the explicit use of XML schema types in the SOAP payload. The default encoding, which is also adopted by the gSOAP stub and skeleton compiler, assumes SOAP encoding. This can be easily changed by using **typedef** definitions in the header file input to the gSOAP compiler. The type name defined by a **typedef** definition corresponds to an XML schema type and may include an optional namespace prefix. For example, the following **typedef** declarations, when part of the header file input to the gSOAP compiler, defines various built-in XML schema types implemented as primitive C/C++ types:

```
// Contents of header file:
...
typedef char *xsd__string; // encode xsd__string value as the xsd:string schema type
typedef char *xsd__anyURI; // encode xsd__anyURI value as the xsd:anyURI schema type
typedef float xsd__float; // encode xsd__float value as the xsd:float schema type
typedef long xsd__int; // encode xsd__int value as the xsd:int schema type
typedef bool xsd__boolean; // encode xsd__boolean value as the xsd:boolean schema type
typedef unsigned long long xsd__positiveInteger; // encode xsd__positiveInteger value as the
xsd:positiveInteger schema type
...
```

This simple mechanism informs the gSOAP compiler to generate serializers and deserializers that explicitly encode and decode the primitive C++ types as built-in primitive XML schema types when the typedefed type is used in the parameter signature of a remote method (or when used nested within structs, classes, and arrays). At the same time, the use of **typedef** does not force any recoding of a C++ client or Web service application as the internal C++ types used by the application are not required to be changed (but still have to be primitive C++ types, see Section 8.2.2 for alternative class implementations of primitive XML schema types which allows for the marshalling of polymorphic primitive SOAP types).

5.1.5 Example

Reconsider the getQuote example, now rewritten with explicit XML schema types to illustrate the effect:

```
// Contents of file "getQuote.h":
typedef char *xsd__string;
typedef float xsd__float;
int ns1__getQuote(xsd__string symbol, xsd__float &Result);
```

This header file is compiled by the gSOAP stub and skeleton compiler and the compiler generates source code for the function soap_call_ns1__getQuote, which is identical to the "old" proxy:

```
int soap_call_ns1__getQuote(struct soap *soap, char *URL, char *action, char *symbol, float
&Result);
```

The client application does not need to be rewritten and can still call the proxy using the "old" parameter signature. In contrast to the previous implementation of the stub however, the encoding and decoding of the data types by the stub has been changed to explicitly use the schema types.

For example, when the client application calls the proxy, the proxy produces a SOAP request with `xsd:string`:

```
...
<SOAP-ENV:Body>
<ns1:getQuote><symbol xsi:type="xsd:string">AOL</symbol>
</ns1:getQuote>
</SOAP-ENV:Body>
...
```

The service response is:

```
...
<soap:Body>
<n:getQuoteResponse xmlns:n='urn:xmethods-delayed-quotes'>
<Result xsi:type='xsd:float'>41.81</Result>
</n:getQuoteResponse>
</soap:Body>
...
```

The validation of this service response by the stub routine takes place by matching the namespace URIs that are bound to the `xsd` namespace prefix. The stub also expects the `getQuoteResponse` element to be associated with namespace URI `urn:xmethods-delayed-quotes` through the binding of the namespace prefix `ns1` in the namespace mapping table. The service response uses namespace prefix `n` for the `getQuoteResponse` element. This namespace prefix is bound to the same namespace URI `urn:xmethods-delayed-quotes` and therefore the service response is assumed to be valid. The response is rejected and a SOAP fault is generated if the namespace URIs do not match.

5.1.6 How to Change the Response Element Name

There is no explicit standard convention for the response element name in SOAP, although it is recommended that the response element name is the method name ending with "`Response`". For example, the response element of `getQuote` is `getQuoteResponse`.

The response element name can be specified explicitly using a **struct** or **class** declaration in the header file. The **struct** or **class** name represents the SOAP response element name used by the service. Consequently, the output parameter of the remote method must be declared as a field of the **struct** or **class**. The use of a **struct** or a **class** for the service response is fully SOAP 1.1 compliant. In fact, the absence of a **struct** or **class** indicates to the gSOAP compiler to automatically generate a struct for the response which is internally used by a stub.

5.1.7 Example

Reconsider the getQuote remote method specification which can be rewritten with an explicit declaration of a SOAP response element as follows:

```
// Contents of "getQuote.h":
typedef char *xsd__string;
typedef float xsd__float;
struct ns1__getQuoteResponse {xsd__float Result;};
int ns1__getQuote(xsd__string symbol, struct ns1__getQuoteResponse &r);
```

The SOAP request is the same as before:

```
...
<SOAP-ENV:Body>
<ns1:getQuote><symbol xsi:type="xsd:string">AOL</symbol>
</ns1:getQuote>
</SOAP-ENV:Body>
...
```

The difference is that the service response is required to match the specified getQuoteResponse name and its namespace URI:

```
...
<soap:Body>
<n:getQuoteResponse xmlns:n='urn:xmethods-delayed-quotes'>
<Result xsi:type='xsd:float'>41.81</Result>
</n:getQuoteResponse>
</soap:Body>
...
```

This use of a **struct** or **class** enables the adaptation of the default SOAP response element name and/or namespace URI when required.

Note that the **struct** (or **class**) declaration may appear within the function prototype declaration. For example:

```
// Contents of "getQuote.h":
typedef char *xsd__string;
typedef float xsd__float;
int ns1__getQuote(xsd__string symbol, struct ns1__getQuoteResponse {xsd__float Result;} &r);
```

This example combines the declaration of the response element of the remote method with the function prototype of the remote method.

5.1.8 How to Specify Multiple Output Parameters

The gSOAP stub and skeleton compiler uses the convention that the **single output parameter** of a remote method is the **last parameter** of the function prototype declaration in a header file. All other parameters are considered input parameters of the remote method. To specify a remote method with **multiple output parameters**, a **struct** or **class** must be declared for the remote method response, see also 5.1.6. The fields of the **struct** or **class** are the output parameters of the remote method. Both the order of the input parameters in the function prototype and the order of the output parameters (the fields in the **struct** or **class**) is not significant. However, the SOAP 1.1 specification states that input and output parameters may be treated as having anonymous parameter names which requires a particular ordering, see Section 5.1.12.

5.1.9 Example

As an example, consider a hypothetical remote method getNames with a single input parameter SSN and two output parameters first and last. This can be specified as:

```
// Contents of file "getNames.h":
int ns3__getNames(char *SSN, struct ns3__getNamesResponse {char *first; char *last;} &r);
```

The gSOAP stub and skeleton compiler takes this header file as input and generates source code for the function soap_call_ns3__getNames. When invoked by a client application, the proxy produces the SOAP request:

```
...
<SOAP-ENV:Envelope ...  xmlns:ns3="urn:names" ...>
...
<ns3:getNames>
<SSN>999 99 9999</SSN>
</ns3:getNames>
...
```

The response by a SOAP service could be:

```
...
<m:getNamesResponse xmlns:m="urn:names">
<first>John</first>
<last>Doe</last>
</m:getNamesResponse>
...
```

where `first` and `last` are the output parameters of the getNames remote method of the service.

As another example, consider a remote method copy with an input parameter and an output parameter with identical parameter names (this is not prohibited by the SOAP 1.1 protocol). This can be specified as well using a response **struct**:

```
// Contente of file "copy.h":
int X_rox__copy_name(char *name, struct X_rox__copy_nameResponse {char *name;} &r);
```

The use of a **struct** or **class** for the remote method response enables the declaration of remote methods that have parameters that are passed both as input and output parameters.

The gSOAP compiler takes the copy.h header file as input and generates the soap_call_X_rox__copy_name proxy. When invoked by a client application, the proxy produces the SOAP request:

```
...
<SOAP-ENV:Envelope ...  xmlns:X-rox="urn:copy" ...>
...
<X-rox:copy-name>
<name>SOAP</name>
</X-rox:copy-name>
...
```

The response by a SOAP copy service could be something like:

```
...
<m:copy-nameResponse xmlns:m="urn:copy">
<name>SOAP</name>
</m:copy-nameResponse>
...
```

The name will be parsed and decoded by the proxy and returned in the name field of the **struct** X_rox__copy_nameResponse &r parameter.

5.1.10 How to Specify Output Parameters With Complex Data Types

If the output parameter of a remote method is a complex data type such as a **struct** or **class** it is necessary to specify the response element of the remote method as a **struct** or **class at all times**. Otherwise, the output parameter will be considered the response element (!), because of the response element specification convention used by gSOAP, as discussed in 5.1.6.

5.1.11 Example

This is is best illustrated with an example. The Flighttracker service by ObjectSpace provides real time flight information for flights in the air. It requires an airline code and flight number as parameters, see `http://xmethods.com/detail.html?id=86` for details. The remote method name is getFlightInfo and the method has two string parameters: the airline code and flight number, both of which must be encoded as `xsd:string` types. The method returns a getFlightResponse response element with a return output parameter that is of complex type FlightInfo. The type FlightInfo is represented by a **class** in the header file, whose field names correspond to the FlightInfo accessors:

```
// Contents of file "flight.h":
typedef char *xsd__string;
class ns2__FlightInfo
{
  public:
  xsd__string airline;
  xsd__string flightNumber;
  xsd__string altitude;
  xsd__string currentLocation;
  xsd__string equipment;
  xsd__string speed;
};
struct ns1__getFlightInfoResponse {ns2__FlightInfo return_;};
int ns1__getFlightInfo(xsd__string param1, xsd__string param2, struct ns1__getFlightInfoResponse
&r);
```

The response element ns1__getFlightInfoResponse is explicitly declared and it has one field: return_ of type ns2__FlightInfo. Note that return_ has a trailing underscore to avoid a name clash with the **return** keyword, see Section 7.3 for details on the translation of C++ identifiers to XML element names.

The SOAP C++ stub and skeleton compiler generates the soap_call_ns1__getFlightInfo proxy. Here is an example fragment of a client application that uses this proxy to request flight information:

```
struct soap soap;
...
soap_init(&soap);
...
soap_call_ns1__getFlightInfo(&soap, "testvger.objectspace.com/soap/servlet/rpcrouter",
  "urn:galdemo:flighttracker", "UAL", "184", r);
...
struct Namespace namespaces[] =
{
  {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
  {"SOAP-ENC","http://schemas.xmlsoap.org/soap/encoding/"},
  {"xsi", "http://www.w3.org/1999/XMLSchema-instance"},
  {"xsd", "http://www.w3.org/1999/XMLSchema"},
  {"ns1", "urn:galdemo:flighttracker"},
  {"ns2", "http://galdemo.flighttracker.com"},
  {NULL, NULL}
};
```

When invoked by a client application, the proxy produces the SOAP request:

```
POST /soap/servlet/rpcrouter HTTP/1.1
Host: testvger.objectspace.com
Content-Type: text/xml
Content-Length: 634
SOAPAction: "urn:galdemo:flighttracker"

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
  xmlns:ns1="urn:galdemo:flighttracker"
  xmlns:ns2="http://galdemo.flighttracker.com"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<ns1:getFlightInfo xsi:type="ns1:getFlightInfo">
<param1 xsi:type="xsd:string">UAL</param1>
<param2 xsi:type="xsd:string">184</param2>
```

```
</ns1:getFlightInfo>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The Flighttracker service responds with:

```
HTTP/1.1 200 ok
Date: Thu, 30 Aug 2001 00:34:17 GMT
Server: IBM_HTTP_Server/1.3.12.3 Apache/1.3.12 (Win32)
Set-Cookie: sesessionid=2GFVTOGC30D0LGRGU2L4HFA;Path=/
Cache-Control: no-cache="set-cookie,set-cookie2"
Expires: Thu, 01 Dec 1994 16:00:00 GMT
Content-Length: 861
Content-Type: text/xml; charset=utf-8
Content-Language: en

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:getFlightInfoResponse xmlns:ns1="urn:galdemo:flighttracker"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xmlns:ns2="http://galdemo.flighttracker.com" xsi:type="ns2:FlightInfo">
<equipment xsi:type="xsd:string">A320</equipment>
<airline xsi:type="xsd:string">UAL</airline>
<currentLocation xsi:type="xsd:string">188 mi W of Lincoln, NE</currentLocation>
<altitude xsi:type="xsd:string">37000</altitude>
<speed xsi:type="xsd:string">497</speed>
<flightNumber xsi:type="xsd:string">184</flightNumber>
</return>
</ns1:getFlightInfoResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The proxy returns the service response in variable r of type **struct** ns1__getFlightInfoResponse and this information can be displayed by the client application with the following code fragment:

```
cout << r.return_.equipment << " flight " << r.return_.airline << r.return_.flightNumber
  << " traveling " << r.return_.speed << " mph " << " at " << r.return_.altitude
  << " ft, is located " << r.return_.currentLocation ¡¡ endl;
```

This code displays the service response as:

```
A320 flight UAL184 traveling 497 mph at 37000 ft, is located 188 mi W of Lincoln,
NE
```

Note: the flight tracker service is no longer available since 9/11/2001.

5.1.12 How to Specify Anonymous Parameter Names

The SOAP 1.1 protocol allows parameter names to be anonymous. That is, the name(s) of the output parameters of a remote method are not strictly required to match a client's view of the parameters names. Also, the input parameter names of a remote method are not striclty required to match a service's view of the parameter names. Although this convention is likely to be deprecated in SOAP 1.2, the gSOAP compiler can generate stub and skeleton routines that support anonymous parameters. To make parameter names anonymous on the receiving side (client or service), the parameter names should start with an underscore (_) in the function prototype in the header file.

For example:

```
// Contents of "getQuote.h":
typedef char *xsd__string;
typedef float xsd__float;
int ns1__getQuote(xsd__string symbol, &_return);
```

Or, alternatively with a response **struct**:

```
// Contents of "getQuote.h":
typedef char *xsd__string;
typedef float xsd__float;
struct ns1__getQuoteResponse {xsd__float _return;};
int ns1__getQuote(xsd__string symbol, struct ns1__getQuoteResponse &r);
```

In this example, _return is an anonymous output parameter. As a consequence, the service response to a request made by a client created with gSOAP using this header file specification may include any name for the output parameter in the SOAP payload. The input parameters may also be anonymous. This affects the implementation of Web services in gSOAP and the matching of parameter names by the service.

Caution: when anonymous parameter names are used, the order of the parameters in the function prototype of a remote method is significant.

5.1.13 How to Specify a Method with No Input Parameters

To specify a remote method that has no input parameters, just provide a function prototype with one parameter which is the output parameter. However, some C/C++ compilers (notably Visual C++^{TM}) will not compile and complain about an empty **struct**. This **struct** is generated by gSOAP to contain the SOAP request message. To fix this, provide one input parameter of type **void*** (gSOAP can not serialize void* data). For example:

```
struct ns3__SOAPService
{
  public:
  int ID;
  char *name;
  char *owner;
  char *description;
  char *homepageURL;
  char *endpoint;
  char *SOAPAction;
  char *methodNamespaceURI;
  char *serviceStatus;
  char *methodName;
```

```
    char *dateCreated;
    char *downloadURL;
    char *wsdlURL;
    char *instructions;
    char *contactEmail;
    char *serverImplementation;
};
struct ArrayOfSOAPService {struct ns3__SOAPService *__ptr; int __size;};
int ns__getAllSOAPServices(void *_, struct ArrayOfSOAPService &_return);
```

The ns__getAllSOAPServices method has one **void*** input parameter which is ignored by the serializer to produce the request message. To call the proxy, use NULL as the actual input parameter value.

Most C/C++ compilers allow empty **struct**s and therefore the **void*** parameter is not required.

5.2 How to Use the gSOAP Stub and Skeleton Compiler to Build SOAP Web Services

The gSOAP stub and skeleton compiler generates **skeleton** routines in C++ source form for each of the remote methods specified as function prototypes in the header file processed by the gSOAP compiler. The skeleton routines can be readily used to implement the remote methods in a new SOAP Web service. The compound data types used by the input and output parameters of SOAP remote methods must be declared in the header file, such as structs, classes, arrays, and pointer-based data structures (graphs) that are used as the data types of the parameters of a remote method. The gSOAP compiler automatically generates serializers and deserializers for the data types to enable the generated skeleton routines to encode and decode the contents of the parameters of the remote methods. The gSOAP compiler also generates a remote method request dispatcher routine that will serve requests by calling the appropriate skeleton when the SOAP service application is installed as a CGI application on a Web server.

5.2.1 Example

The following example specifies three remote methods to be implemented by a new SOAP Web service:

```
// Contents of file "calc.h":
typedef double xsd__double;
int ns__add(xsd__double a, xsd__double b, xsd__double &result);
int ns__sub(xsd__double a, xsd__double b, xsd__double &result);
int ns__sqrt(xsd__double a, xsd__double &result);
```

The add and sub methods are intended to add and subtract two double floating point numbers stored in input parameters a and b and should return the result of the operation in the result output parameter. The qsrt method is intended to take the square root of input parameter a and to return the result in the output parameter result. The xsd__double type is recognized by the gSOAP compiler as the `xsd:double` XML schema data type. The use of **typedef** is a convenient way to associate primitive C types with primitive XML schema data types.

To generate the skeleton routines, the gSOAP compiler is invoked from the command line with:

```
soapcpp2 calc.h
```

The compiler generates the skeleton routines for the add, sub, and sqrt remote methods specified in the calc.h header file. The skeleton routines are respectively, soap_serve_ns__add, soap_serve_ns__sub, and soap_serve_ns__sqrt and saved in the file soapServer.cpp. The generated file soapC.cpp contains serializers and deserializers for the skeleton. The compiler also generates a service dispatcher: the soap_serve function handles client requests on the standard input stream and dispatches the remote method requests to the appropriate skeletons to serve the requests. The skeleton in turn calls the remote method implementation function. The function prototype of the remote method implementation function is specified in the header file that is input to the gSOAP compiler.

Here is an example Calculator service application that uses the generated soap_serve routine to handle client requests:

```
// Contents of file "calc.cpp":
#include "soapH.h"
#include <math.h> // for sqrt()
main()
{
  soap_serve(soap_new()); // use the remote method request dispatcher
}
// Implementation of the "add" remote method:
int ns__add(struct soap *soap, double a, double b, double &result)
{
  result = a + b;
  return SOAP_OK;
}
// Implementation of the "sub" remote method:
int ns__sub(struct soap *soap, double a, double b, double &result)
{
  result = a - b;
  return SOAP_OK;
}
// Implementation of the "sqrt" remote method:
int ns__sqrt(struct soap *soap, double a, double &result);
{
  if (a >= 0)
  {
    result = sqrt(a);
    return SOAP_OK;
  }
  else
  {
    soap_fault(soap); // allocate space for fault (if necessary)
    soap->fault->faultstring = "Square root of negative number";
    soap->fault->detail = "I can only take the square root of a non-negative number";
    return SOAP_FAULT;
  }
}
// As always, a namespace mapping table is needed:
struct Namespace namespaces[] =
{  // {"ns-prefix", "ns-name"}
  {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
  {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"},
  {"xsi", "http://www.w3.org/1999/XMLSchema-instance"},
  {"xsd", "http://www.w3.org/1999/XMLSchema"},
  {"ns", "urn:simple-calc"}, // bind "ns" namespace prefix
  {NULL, NULL}
};
```

Note that the remote methods have an extra input parameter which is a pointer to the gSOAP runtime environment. The implementation of the remote methods MUST return a SOAP error code. The code SOAP_OK denotes success, while SOAP_FAULT denotes an exception with details that can be defined by the user. The exception description can be assigned to the soap->fault->faultstring string and details can be assigned to the soap->fault->detail string. The soap_fault function will allocate a fault struct. The fault exception will be passed on to the client of this service.

This service application can be readily installed as a CGI application. The service description would be:

Endpoint URL:	the URL of the CGI application
SOAP action:	"" (2 quotes)
Remote method namespace:	`urn:simple-calc`
Remote method name:	`add`
Input parameters:	`a` of type `xsd:double` and `b` of type `xsd:double`
Output parameter:	`result` of type `xsd:double`
Remote method name:	`sub`
Input parameters:	`a` of type `xsd:double` and `b` of type `xsd:double`
Output parameter:	`result` of type `xsd:double`
Remote method name:	`sqrt`
Input parameter:	`a` of type `xsd:double`
Output parameter:	`result` of type `xsd:double` or a SOAP Fault

The soapcpp2 compile generates a WSDL file for this service, see Section 5.2.5.

Unless the CGI application inspects and checks the environment variable SOAPAction which contains the SOAP action request by a client, the SOAP action is ignored by the CGI application. SOAP actions are specific to the SOAP protocol and provide a means for routing requests and for security reasons (e.g. firewall software can inspect SOAP action headers to grant or deny the SOAP request. Note that this requires the SOAP service to check the SOAP action header as well to match it with the remote method.)

The header file input to the gSOAP compiler does not need to be modified to generate client stubs for accessing this service. Client applications can be developed by using the same header file as for which the service application was developed. For example, the soap_call_ns__add proxy is available from the soapClient.cpp file after invoking the gSOAP compiler on the calc.h header file. As a result, client and service applications can be developed without the need to know the details of the SOAP encoding used.

5.2.2 How to Create a Stand-Alone Service

The deployment of a Web service as a CGI application is an easy means to provide your service on the Internet. Services can also run as stand-alone services on intranets where client-service interactions are not blocked by firewalls.

To create a stand-alone service, only the main routine of the service needs to be modified. Instead of just calling the soap_serve routine, the main routine is changed into:

```
int main()
{
   struct soap soap;
   int m, s; // master and slave sockets
   soap_init(&soap);
   m = soap_bind(&soap, "machine.cs.fsu.edu", 18083, 100);
   if (m < 0)
      soap_print_fault(&soap, stderr);
   else
   {
      fprintf(stderr, "Socket connection successful: master socket = %d\n", m);
      for (int i = 1; ; i++)
      {
         s = soap_accept(&soap);
         if (s < 0)
         {
            soap_print_fault(&soap, stderr);
            break;
         }
         fprintf(stderr, "%d: accepted connection from IP = %d.%d.%d.%d socket = %d", i,
            (soap.ip<<24)&0xFF, (soap.ip<<16)&0xFF, (soap.ip<<8)&0xFF, soap.ip&0xFF, s);
         soap_serve(&soap); // process RPC skeletons
         fprintf(stderr, "request served\n");
         soap_destroy(&soap); // clean up class instances
         soap_end(&soap); // clean up everything and close socket
      }
   }
   soap_done(&soap); // close master socket
}
```

The functions used are:

Function	**Description**
soap_init(**struct** soap *soap)	Initializes gSOAP runtime environment (required once)
soap_bind(**struct** soap *soap, **char** *host, **int** port, **int** backlog)	Returns master socket (backlog = max. queue size for requests). When host==NULL: host is the machine on which the service runs
soap_accept(**struct** soap *soap)	Returns slave socket
soap_end(**struct** soap *soap)	Clean up deserialized data (except class instances) and temporary data
soap_free(**struct** soap *soap)	Clean up temporary data only
soap_destroy(**struct** soap *soap)	Clean up deserialized class instances
soap_done(**struct** soap *soap)	Close master socket

The *host* name in soap_bind may be NULL to indicate that the current host should be used.

The soap.accept_timeout attribute of the gSOAP run-time environment specifies the timeout value for a non-blocking soap_accept(&soap) call. See Section 12.10 for more details on timeout management.

See Section 6.7 for more details on memory management.

A client application connects to this stand-alone service with the endpoint machine.cs.fsu.edu:18083. A client may use the http:// prefix. When absent, no HTTP header is send and no HTTP-based information will be communicated to the service.

5.2.3 How to Create a Multi-Threaded Stand-Alone Service

Multi-threading a Web Service is essential when the response times for handling requests by the service are (potentially) long. In case of long response times, the latencies introduced by the unrelated requests may become prohibitive for a successful deployment of a stand-alone service.

gSOAP 2.0 is thread safe and allows the implementation of multi-threaded stand-alone services. Multiple threads can be used to handle requests.

Here is an example of a multi-threaded Web Service:

```
#include "soapH.h"
#include <pthread.h>
#define BACKLOG (100) // Max. request backlog
#define MAX_THR (8) // Max. threads to serve requests
int main(int argc, char **argv)
{
   structsoap soap;
   soap_init(&soap);
   if (argc < 3) // no args: assume this is a CGI application
   {
      soap_serve(&soap); // serve request, one thread, CGI style
      soap_end(&soap); // cleanup
   }
   else
   {
      struct soap *soap_thr[MAX_THR]; // each thread needs a runtime environment
      pthread_t tid[MAX_THR];
      char *host = argv[1];
      int port = atoi(argv[2]);
      int m, s, i;
      m = soap_bind(&soap, host, port, BACKLOG);
      if (m < 0)
         exit(-1);
      fprintf(stderr, "Socket connection successful %d\n", m);
      for (i = 0; i < MAX_THR; i++)
         soap_thr[i] = NULL;
      for (;;)
      {
         for (i = 0; i < MAX_THR; i++)
         {
            s = soap_accept(&soap);
            if (s < 0)
               break;
            fprintf(stderr, "Thread %d accepts socket %d connection from IP %d.%d.%d.%d\n",
i, s, (soap.ip>>24)&0xFF, (soap.ip>>16)&0xFF, (soap.ip>>8)&0xFF, soap.ip&0xFF);
            if (!soap_thr[i]) // first time around
            {
               soap_thr[i] = soap_new();
               if (!soap_thr[i])
               exit(-1); // could not allocate
            }
            else // recycle soap environment
            {
               pthread_join(tid[i], NULL);
               fprintf(stderr, "Thread %d completed\n", i);
               soap_end(soap_thr[i]); // deallocate data of old thread
            }
            soap_thr[i]->socket = s;
            pthread_create(&tid[i], NULL, (void*(*)(void*))soap_serve, (void*)soap_thr[i]);
         }
      }
   }
   return 0;
}
```

The example illustrates the use of threads to improve the quality of service by handling new requests in separate threads. Each thread needs a separate runtime environment. The example above requires threads to synchronize at some point, so runaway processes can be halted (not shown in the code). The next example detaches threads. No attempt is made to synchronize threads. Runaway threads will consume resources.

```
#include "soapH.h"
#include <pthread.h>
#define BACKLOG (100) // Max. request backlog
int main(int argc, char **argv)
{
   struct soap soap;
   soap_init(&soap);
   if (argc < 3) // no args: assume this is a CGI application
   {
      soap_serve(&soap); // serve request, one thread, CGI style
      soap_end(&soap); // cleanup
   }
   else
   {
      void *process_request(void*);
      struct soap *tsoap;
      pthread_t tid;
      char *host = argv[1];
      int port = atoi(argv[2]);
      int m, s;
      m = soap_bind(&soap, host, port, BACKLOG);
      if (m < 0)
         exit(-1);
      fprintf(stderr, "Socket connection successful %d\n", m);
      for (;;)
      {
         s = soap_accept(&soap);
         if (s < 0)
            break;
         fprintf(stderr, "Thread %d accepts socket %d connection from IP %d.%d.%d.%d\n",
i, s, (soap.ip>>24)&0xFF, (soap.ip>>16)&0xFF, (soap.ip>>8)&0xFF, soap.ip&0xFF);
         tsoap = soap_new();
         if (!tsoap)
            break;
         tsoap->socket = s;
         pthread_create(&tid, NULL, (void*(*)(void*))process_request, (void*)tsoap);
      }
   }
   return 0;
}
void *process_request(void *soap)
{
   pthread_detach(pthread_self());
   soap_serve((struct soap*)soap);
   soap_end((struct soap*)soap);
   free(soap);
   return NULL;
}
```

For clean termination of the server, the master socket can be closed and callbacks removed with `soap_done(struct soap *soap)`.

5.2.4 Some Web Service Implementation Issues

The same client header file specification issues apply to the specification and implementation of a SOAP Web service. Refer to

- 5.1.2 for namespace considerations.
- 5.1.4 for an explanation on how to change the encoding of the primitive types.
- 5.1.6 for a discussion on how the response element format can be controlled.
- 5.1.8 for details on how to pass multiple output parameters from a remote method.
- 5.1.10 for passing complex data types as output parameters.
- 5.1.12 for anonymizing the input and output parameter names.

5.2.5 How to Generate WSDL Service Descriptions

The gSOAP stub and skeleton compiler `soapcpp2` generates WSDL (Web Service Description Language) service descriptions and XML schema files when processing a header file. The compiler produces one WSDL file for a set of remote methods. The names of the function prototypes of the remote methods must use the same namespace prefix and the namespace prefix is used to name the WSDL file. If multiple namespace prefixes are used to define remote methods, multiple WSDL files will be created and each file describes the set of remote methods belonging to a namespace prefix.

To publish the WSDL service description, the `%{}%`-patterns that appear in the generated WSDL file have to be filled in with the following information:

Replace	With
%{Service}%	the file name of the CGI service application (without a file name extension)
%{URL}%	the endpoint URL of the service (without the CGI file name)
%{URI}%	the namespace URI of the service (can be the same as the URL)

This information can also be provided in the header file which will then be automatically incorporated in the WSDL file, see advanced features Section 12.2.

In addition to the generation of the `ns.wsdl` file, a file with a namespace mapping table is generated by the gSOAP compiler. An example mapping table is shown below:

```
struct Namespace namespaces[] =
{
  {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
  {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"},
  {"xsi", "http://www.w3.org/2001/XMLSchema-instance", "http://www.w3.org/*/XMLSchema-
instance"},
  {"xsd", "http://www.w3.org/2001/XMLSchema", "http://www.w3.org/*/XMLSchema"},
  {"ns", "%{URI}%"},
  {NULL, NULL}
};
```

After replacing the %{}% patterns in the namespace mapping table file, this file can be incorporated in the client/service application, see Section 7.4 for details on namespace mapping tables.

To deploy a Web service, copy the compiled CGI service application to the designated CGI directory of your Web server. Make sure the file permissions are set right (`chmod 755 calc.cgi` for Unix/Linux). You can then publish the WSDL file on the Web.

The gSOAP compiler also generates XML schema files for all C/C++ complex types (e.g. **struct**s and **class**es) when declared with a namespace prefix. These files are named `ns.xsd`, where `ns` is the namespace prefix used in the declaration of the complex type. The XML schema files do not have to be published as the WSDL file already contains the appropriate XML schema types.

5.2.6 Example

For example, suppose the following methods are defined in the header file:

```
typedef double xsd__double;
int ns__add(xsd__double a, xsd__double b, xsd__double &result);
int ns__sub(xsd__double a, xsd__double b, xsd__double &result);
int ns__sqrt(xsd__double a, xsd__double &result);
```

Then, one WSDL file will be created with the file name `ns.wsdl` that describes all three remote methods:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="%{Service}%"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  targetNamespace="%{URL}%/%{Service}%.wsdl"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:WSDL="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
  xmlns:tns="%{URL}%/%{Service}%.wsdl"
  xmlns:ns="%{URL}%/ns.xsd">
<types>
  <schema
    xmlns="http://www.w3.org/2000/10/XMLSchema"
    targetNamespace="%{URL}%/ns.xsd"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
    <complexType name="addResponse">
      <all>
        <element name="result" type="double" minOccurs="0" maxOccurs="1"/>
      </all>
      <anyAttribute namespace="##other"/>
    </complexType>
    <complexType name="subResponse">
      <all>
        <element name="result" type="double" minOccurs="0" maxOccurs="1"/>
      </all>
      <anyAttribute namespace="##other"/>
    </complexType>
    <complexType name="sqrtResponse">
      <all>
        <element name="result" type="double" minOccurs="0" maxOccurs="1"/>
      </all>
      <anyAttribute namespace="##other"/>
    </complexType>
  </schema>
</types>
<message name="addRequest">
  <part name="a" type="xsd:double"/>
  <part name="b" type="xsd:double"/>
</message>
<message name="addResponse">
  <part name="result" type="xsd:double"/>
</message>
<message name="subRequest">
  <part name="a" type="xsd:double"/>
  <part name="b" type="xsd:double"/>
</message>
<message name="subResponse">
  <part name="result" type="xsd:double"/>
</message>
<message name="sqrtRequest">
  <part name="a" type="xsd:double"/>
</message>
<message name="sqrtResponse">
  <part name="result" type="xsd:double"/>
</message>
<portType name="%{Service}%PortType">
```

```
  <operation name="add">
    <input message="tns:addRequest"/>
    <output message="tns:addResponse"/>
  </operation>
  <operation name="sub">
    <input message="tns:subRequest"/>
    <output message="tns:subResponse"/>
  </operation>
  <operation name="sqrt">
    <input message="tns:sqrtRequest"/>
    <output message="tns:sqrtResponse"/>
  </operation>
</portType>
<binding name="%{Service}%Binding" type="tns:%{Service}%PortType">
  <SOAP:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="add">
    <SOAP:operation soapAction="%{URI}%#add"/>
    <input>
      <SOAP:body use="encoded" namespace="%{URI}%"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output>
      <SOAP:body use="encoded" namespace="%{URI}%"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>
  <operation name="sub">
    <SOAP:operation soapAction="%{URI}%#sub"/>
    <input>
      <SOAP:body use="encoded" namespace="%{URI}%"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output>
      <SOAP:body use="encoded" namespace="%{URI}%"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>
  <operation name="sqrt">
    <SOAP:operation soapAction="%{URI}%#sqrt"/>
    <input>
      <SOAP:body use="encoded" namespace="%{URI}%"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output>
      <SOAP:body use="encoded" namespace="%{URI}%"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>
</binding>
<service name="%{Service}%">
  <port name="%{Service}%Port" binding="tns:%{Service}%Binding">
    <SOAP:address location="%{URL}%/%{Service}%.cgi"/>
  </port>
```

```
</service>
</definitions>
```

The service name of the calculator service could be `calc` (so the file name of the CGI service application is `calc.cgi`), the URL could be `http://www.mycalc.com`, and the namespace URI could be `http://www.mycalc.com` (the URI must be unique, if possible, and the URL uniquelly identifies an organization). According to this, the following modifications to the generated WSDL file have to be made:

Replace	**With**
%{Service}%	`calc`
%{URL}%	`http://www.mycalc.com`
%{URI}%	`http://www.mycalc.com`

5.2.7 How to Import WSDL Service Descriptions

The creation of SOAP Web Service clients from a WSDL service description is a two-step process.

First, execute java wsdlcpp file.wsdl which generates the a header file file.h and a C-source file file.c with an example client program template. Modify the client program template to your needs.

Second, the header file file.h is to be processed by the gSOAP compiler by executing soapcpp2 file.h. This creates the C-source files to build a client application, see 5.1.

The following limitations are specific to the WSDL importer tool. The limitations are not general limitations of the gSOAP toolkit and the gSOAP stub and skeleton compiler. Future releases of the WSDL import tool will address these limitations.

- No `<import>` (WSDL must be self-contained)
- No support for SOAP Header and Fault messages If Header processing is required, this will need to be added by hand to the generated header file.
- To ensure compatibility to C, the current WSDL importer generates **struct** declarations. These can be changed into **class** declarations in the generated header file when necessary.

5.2.8 How to Specify minOccurs and maxOccurs Schema Attributes

By default, gSOAP generates WSDL and schemas with `minOccurs=1` and `maxOccurs=1` for non-array types, and `minOccurs=0` and `maxOccurs=unbounded` for array types. The `minOccurs` and `maxOccurs` attribute values of fields in **struct** and **class** types are specified as

Type fieldname [minOccurs[:maxOccurs]] [= value]

The minOccurs and maxOccurs values must be integer literals.

For example

```
struct ns__MyRecord
{
  int n;
```

```
    int m 0;
    int __size 0:10;
    int *item;
}
```

gSOAP generates:

```
<complexType name="MyRecord">
  <all>
    <element name="n" type="xsd:int" minOccurs="1" maxOccurs="1"/>
    <element name="m" type="xsd:int" minOccurs="0" maxOccurs="1"/>
    <element name="item" type="xsd:int" minOccurs="0" maxOccurs="10"/>
  </all>
</complexType>
```

5.2.9 Combining a Client and Service into a Peer Application

This is a more sophisticated example that combines the functionality of two Web services into one new SOAP Web service. The service provides a currency-converted stock quote. To serve a request, the service in turn requests the stock quote and the currency-exchange rate from two XMethods services.

In addition to being a client of two XMethods services, this service application can also be used as a client of itself to test the implementation. As a client invoked from the command-line, it will return a currency-converted stock quote by connecting to a copy of itself installed as a CGI application on the Web to retrieve the quote after which it will print the quote on the terminal.

The header file input to the gSOAP compiler is given below:

```
// Contents of file "quotex.h":
int ns1__getQuote(char *symbol, float &result); // XMethods delayed stock quote service remote
method
int ns2__getRate(char *country1, char *country2, float &result); // XMethods currency-exchange
service remote method
int ns3__getQuote(char *symbol, char *country, float &result); // the new currency-converted
stock quote service
```

The quotex.cpp client/service application source is:

```
// Contents of file "quotex.cpp":
#include "soapH.h" // include generated proxy and SOAP support
int main(int argc, char **argv)
{
  struct soap soap;
  float q;
  soap_init(&soap);
  if (argc ¡= 2)
    soap_serve();
  else if (soap_call_ns3__getQuote(&soap, "http://www.cs.fsu.edu/~engelen/quotex.cgi",
NULL, argv[1], argv[2], q))
    soap_print_fault(&soap, stderr);
```

```
   else
      printf("\nCompany %s: %f (%s)\n", argv[1], q, argv[2]);
   return 0;
}
int ns3__getQuote(struct soap *soap, char *symbol, char *country, float &result)
{
   float q, r;
   if (soap_call_ns1__getQuote(soap, "http://services.xmethods.net/soap", NULL, symbol, q)
== 0 &&
      soap_call_ns2__getRate(soap, "http://services.xmethods.net/soap", NULL, "us", coun-
try, r) == 0)
   {
      result = q*r;
      return SOAP_OK;
   }
   else
      return SOAP_FAULT; // pass soap fault messages on to the client of this app
}
/* Since this app is a combined client-server, it is put together with
one header file that describes all remote methods. However, as a consequence we
have to implement the methods that are not ours. Since these implementations are
never called (this code is client-side), we can make them dummies as below.
/
int ns1__getQuote(structsoap *soap, char *symbol, float &result)
{ return SOAP_NO_METHOD; } // dummy: will never be called
int ns2__getRate(structsoap *soap, char *country1, char *country2, float &result)
{ return SOAP_NO_METHOD; } // dummy: will never be called

struct Namespace namespaces[] =
{
   {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
   {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"},
   {"xsi", "http://www.w3.org/2001/XMLSchema-instance", "http://www.w3.org/*/XMLSchema-
instance"},
   {"xsd", "http://www.w3.org/2001/XMLSchema", "http://www.w3.org/*/XMLSchema"},
   {"ns1", "urn:xmethods-delayed-quotes"},
   {"ns2", "urn:xmethods-CurrencyExchange"},
   {"ns3", "urn:quotex"},
   {NULL, NULL}
};
```

To compile:

```
soapcpp2 quotex.h
g++ -o quotex.cgi quotex.cpp soapC.cpp soapClient.cpp soapServer.cpp stdsoap2.cpp -lsocket -lxnet
-lnsl -lm
```

Note: under Linux you can omit the `-l` libraries.

The quotex.cgi executable is installed as a CGI application on the Web by copying it in the designated directory specific to your Web server. After this, the executable can also serve to test the service. For example

```
quotex.cgi AOL uk
```

returns the quote of AOL in uk pounds by communicating the request and response quote from the CGI application. See `http://xmethods.com/detail.html?id=5` for details on the currency abbreviations.

When combining clients and service functionalities, it is required to use one header file input to the compiler. As a consequence, however, stubs and skeletons are available for **all** remote methods, while the client part will only use the stubs and the service part will use the skeletons. Thus, dummy implementations of the unused remote methods need to be given which are never called.

Three WSDL files are created by gSOAP: `ns1.wsdl`, `ns2.wsdl`, and `ns3.wsdl`. Only the `ns3.wsdl` file is required to be published as it contains the description of the combined service, while the others are generated as a side-effect (and in case you want to develop these separate services).

5.3 How to Use gSOAP for One-Way SOAP Messaging

The default gSOAP client-server interaction is synchonous: the client waits for the server to respond to the request. gSOAP also supports "one-way" SOAP messaging. SOAP messaging routines are declared as function prototypes, just like remote methods for SOAP RPC. However, the output parameter is a **void** type to indicate the absence of a return value.

For example, the following header file specifies a event message for SOAP messaging:

```
int ns__event(int eventNo, void dummy);
```

The gSOAP stub and skeleton compiler generates the following functions in soapClient.cpp:

```
int soap_send_ns__event(struct soap *soap, const char URL, const char action, int event);
int soap_recv_ns__event(struct soap *soap, struct ns__event *dummy);
```

The soap_send_ns__event function transmits the message to the destination URL by opening a socket and sending the SOAP encoded message. The socket will remain open after the send and has to be closed with soap_closesock(). The open socket connection can also be used to obtain a service response, e.g. with a soap_recv function call.

The soap_recv_ns__event function waits for a SOAP message on the currently open socket (soap.socket) and fills the **struct** ns__event with the ns__event parameters (e.g. **int** eventNo). The **struct** ns__event is automatically created by gSOAP and is a mirror image of the ns__event parameters:

```
struct ns__event
{ int eventNo;
}
```

The gSOAP generated soapServer.cpp code includes a skeleton routine to accept the message. (The skeleton routine does not respond with a SOAP response message.)

```
int soap_serve_ns__event(struct soap *soap);
```

The skeleton routine calls the user-implemented ns__event(**struct** soap *soap, **int** eventNo) routine (note tha absence of the void parameter!).

As usual, the skeleton will be automatically called by the remote method request dispatcher that handles both the remote method requests (RPCs) and messages:

```
main()
{ soap_serve(soap_new());
}
int ns__event(struct soap *soap, int eventNo)
{
  ... // handle event
  return SOAP_OK;
}
```

5.4 How to Separately Use the SOAP Serializers and Deserializers

The gSOAP stub and skeleton compiler generates serializers and deserializers for all user-defined data structures that are specified in the header file input to the compiler. The serializers and deserializers can be found in the generated soapC.cpp file. These serializers and deserializers can be used separately by an application without the need to build a full client or service application. This is useful for applications that need to save or export their data in XML or need to import data in XML format that is possibly saved by other applications.

The following attributres can be set to control the destination and source for serialization and deserialization:

Variable	Description
soap.socket	socket file descriptor for input and output or -1
soap.sendfd	if soap_socket<0, file descriptor for send operations
soap.sendfd	if soap_socket<0, file descriptor for receive operations
soap.buffering	when not zero, a send buffer is used

The following initializing and finalizing functions can be used:

Function	Description
void soap_begin_send(**struct** soap*)	use buffered socket sends when soap.socket≥0
int soap_end_send(**struct** soap*)	flush the buffer
int soap_begin_recv(**struct** soap*)	if an HTTP header is present, parse it first
int soap_end_recv(**struct** soap*)	perform a id/href consistancy check on deserialized data

5.4.1 Serializing a Data Type

To serialize a data type, two functions need to be called to process the data. The first function (soap_serialize) analyzes pointers and determines if multi-references are required to encode the data and if the data contains cycles. The second function (soap_put) generates the SOAP encoding output for that data type.

The function names are specific to a data type. For example, soap_serialize_float(&soap, &d) is called to serialize an **float** value and soap_put_float(&soap, &d, "number", NULL) is called to output the floating point value in SOAP tagged with the name `<number>`. To initialize data, the soap_default function of a data type can be used. For example, soap_default_float(&soap, &d) initializes the float to 0.0. The soap_default functions are useful to initialize complex data types such as arrays, **struct**s, and **class** instances. Note that the soap_default functions do not need the gSOAP runtime environment as a first parameter.

The following table lists the type naming conventions used:

Type	**Type Name**
char*	string
wchar_t*	wstring
char	byte
bool	bool
double	double
int	int
float	float
long	long
LONG64	LONG64 (Win32)
long long	LONG64 (Unix/Linux)
short	short
time_t	time
unsigned char	unsignedByte
unsigned int	unsignedInt
unsigned long	unsignedLong
ULONG64	unsignedLONG64 (Win32)
unsigned long long	unsignedLONG64 (Unix/Linux)
unsigned short	unsignedShort
T[N]	ArrayNOfType where Type is the type name of T
T*	PointerToType where Type is the type name of T
struct Name	Name
class Name	Name
enum Name	Name

Consider for example the following declaration of p as a pointer to a **struct** ns__Person:

```
struct ns__Person { char *name; } *p;
```

To serialize p, its address is passed to the function soap_serialize_PointerTons__Person generated for this type by the gSOAP compiler:

```
soap_serialize_PointerTons__Person(&soap, &p);
```

The **address of** p is passed, so the serializer can determine whether p was already serialized and to discover cycles in graph data structures. To generate the output, the address of p is passed to the function soap_put_PointerTons__Person together with the name of an XML element and an optional type string (to omit a type, use NULL):

```
soap_put_PointerTons__Person(&soap, &p, "ns:element-name", "ns:type-name");
```

This produces:

```
<ns:element-name xmlns:SOAP-ENV="..." xmlns:SOAP-ENC="..." xmlns:ns="..."
  ...  xsi:type="ns:type-name">
<name xsi:type="xsd:string">...</name>
</ns:element-name>
```

The serializer is initialized with the soap_begin function. All temporary data structures and data structures deserialized on the heap are destroyed with the soap_end() function. The soap_free() function can be used to remove the temporary data only and keep the deserialized data on the heap. Temporary data structures are only created if the encoded data uses pointers. Each pointer in the encoded data has an internal hash table entry to determine all multi-reference parts and cyclic parts of the complete data structure.

If more than one data structure is to be serialized and parts of those data structures are shared through pointers, then the soap_serialize functions MUST to be called first before any of the soap_put functions. This is necessary to ensure that multi-reference data shared by the data structures is encoded as multi-reference.

For example, to encode the contents of two variables var1 and var2 the serializers are called before the output routines:

```
T1 var1;
T2 var2;
struct soap soap;
...
soap_init(&soap); // initialize at least once
soap_begin(&soap); // start new serialization phase
soap.enable_embedding = 1; // do not use independent elements
soap_serialize_Type1(&soap, &var1);
soap_serialize_Type2(&soap, &var2);
...
[soap.socket = a_socket_file_descriptor;]
[soap.sendfd = an_output_file_descriptor;]
[soap_begin_send(&soap);] // use buffered socket output
soap_put_Type1(&soap, &var1, "[namespace-prefix:]element-name1", "[namespace-prefix:]type-name1");
soap_put_Type2(&soap, &var2, "[namespace-prefic:]element-name2", "[namespace-prefix:]type-name2");
...
[soap_end_send(&soap);] // flush buffered socket output
soap_end(&soap); // remove temporary data structures
...
```

where Type1 is the type name of T1 and Type2 is the type name of T2 (see table above). The strings [namespace-prefix:]type-name1 and [namespace-prefix:]type-name2 describe the schema types of the elements. Use NULL to omit this type information. The output stream is set by the assignment to soap.sendfd.

For serializing class instances, method invocations MUST be used instead of function calls, for example var.soap_serialize(&soap) and var.soap_put(&soap, "elt", "type"). This ensures that the proper serializers are used for serializing instances of derived classes.

In principle, encoding MAY take place without calling the soap_serialize functions. However, as the following example demonstrates the resulting encoding is not SOAP 1.1 compliant. However, the messages can still be used with gSOAP to save and restore data.

Consider the following **struct**:

```
// Contents of file "tricky.h":
struct Tricky
{
  int *p;
  int n;
  int *q;
};
```

The following fragment initializes the pointer fields p and q to the value of field n:

```
struct soap soap;
struct Tricky X;
X.n = 1;
X.p = &X.n;
X.q = &X.n;
soap_init(&soap);
soap_begin(&soap);
soap_serialize_Tricky(&soap, &X);
soap_put_Tricky(&soap, &X, "Tricky", NULL);
soap_end(&soap); // Clean up temporary data used by the serializer
```

The resulting output is:

```
<Tricky xsi:type="Tricky">
<p href="#2"/> <n xsi:type="int">1</n> <q href="#2"/> <r xsi:type="int">2</r> </Tricky>
<id id="2" xsi:type="int">1</id>
```

which uses an independent element at the end to represent the multi-referenced integer.

To preserve the exact structure of the data, use the setting soap.enable_embedding=1 (see Section 6.6) to serialize multi-referenced data embedded in the structure which assures the preservation of structure but is not SOAP 1.1 compliant. For example, the resulting output is:

```
<Tricky xsi:type="Tricky">
<p href="#2"/> <n id="2" xsi:type="int">1</n> <q href="#2"/> </Tricky>
```

In this case, the XML is self-contained and multi-referenced data is accurately serialized. The gSOAP generated deserializer for this data type will be able to accurately reconstruct the data from the XML (on the heap).

5.4.2 Deserializing a Data Type

To deserialize a data type, its soap_get function is used. The outline of a program that deserializes two variables var1 and var2 is for example:

```
T1 var1;
T2 var2;
struct soap soap;
...
soap_init(&soap); // initialize at least once
soap_begin(&soap); // begin new decoding phase
[soap.recvfd = an_input_stream;]
[soap_begin_recv(&soap);] // if HTTP header is present, parse it
soap_get_Type1(&soap, &var1, "[namespace-prefix:]element-name1", "[namespace-prefix:]type-name1");
soap_get_Type2(&soap, &var2, "[namespace-prefix:]element-name2", "[namespace-prefix:]type-name1");
...
[soap_end_recv(&soap);] // check consistancy of id/hrefs
soap_end(&soap); // remove temporary data, including the decoded data on the heap
```

The strings [namespace-prefix:]type-name1 and [namespace-prefix:]type-name2 are the schema types of the elements and should match the `xsi:type` attribute of the receiving message. To omit the match, use NULL as the type. For class instances, method invocation can be used instead of a function call if the object is already instantiated, i.e. var.soap_get(&soap, "...", "...").

The soap_begin call resets the deserializers. The soap_end call removes the temporary data structures **and** the decoded data that was placed on the heap. Temporary data is created only if the SOAP content includes `id` and `href` attributes. An internal hash table is used by the deserializer to bound the `id` with the `href` names to reconstruct the shape of the data structure.

To remove temporary data while retaining the deserilzed data on the heap, the function soap_free should be called instead of soap_end.

5.4.3 Example

As an example, consider the following data type declarations:

```
// Contents of file "person.h":
typedef char *xsd__string;
typedef char *xsd__Name;
typedef unsigned int xsd__unsignedInt;
enum ns__Gender {male, female};
class ns__Address
{
  public:
  xsd__string street;
  xsd__unsignedInt number;
  xsd__string city;
};
class ns__Person
{
  public:
  xsd__Name name;
  enum ns__Gender gender;
  ns__Address address;
  ns__Person *mother;
  ns__Person *father;
};
```

The following program uses these data types to store a person named "John" living at Dowling st. 10 in Londen. He has a mother "Mary" and a father "Stuart". After initialization, the class instance for "John" is serialized and encoded in SOAP to the standard output stream:

```
// Contents of file "person.cpp":
#include "soapH.h"
int main()
{
  struct soap soap;
  ns__Person mother, father, john;
  soap.enable_embedding = 1; // see 6.6
  mother.name = "Mary";
  mother.gender = female;
```

```
   mother.address.street = "Dowling st.";
   mother.address.number = 10;
   mother.address.city = "London";
   mother.mother = NULL;
   mother.father = NULL;
   father.name = "Stuart";
   father.gender = male;
   father.address.street = "Main st.";
   father.address.number = 5;
   father.address.city = "London";
   father.mother = NULL;
   father.father = NULL;
   john.name = "John";
   john.gender = male;
   john.address = mother.address;
   john.mother = &mother;
   john.father = &father;
   soap_init(&soap);
   soap_begin(&soap);
   john.soap_serialize(&soap);
   john.soap_put(&soap, "johnnie", NULL);
   soap_end(&soap);
}
struct Namespace namespaces[] =
{
   {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
   {"SOAP-ENC","http://schemas.xmlsoap.org/soap/encoding/"},
   {"xsi", "http://www.w3.org/1999/XMLSchema-instance"},
   {"xsd", "http://www.w3.org/1999/XMLSchema"},
   {"ns", "urn:person"}, // Namespace URI of the "Person" data type
   {NULL, NULL}
};
```

The header file is processed and the application compiled on Linux/Unix with:

```
soapcpp2 person.h
g++ -o person person.cpp soapC.cpp stdsoap2.cpp -lsocket -lxnet -lnsl -lm
```

(Depending on your system configuration, the libraries libsocket.a, libxnet.a, libnsl.a are required. Compiling on Linux typically does not require the inclusion of those libraries.)

Running the person application results in the SOAP output:

```
<johnnie xsi:type="ns:Person" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
  xmlns:ns="urn:person"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<name xsi:type="xsd:Name">John</name>
<gender xsi:type="ns:Gender">male</gender>
<address xsi:type="ns:Address">
<street id="3" xsi:type="xsd:string">Dowling st.</street>
```

```
<number xsi:type="unsignedInt">10</number>
<city id="4" xsi:type="xsd:string">London</city>
</address>
<mother xsi:type="ns:Person">
<name xsi:type="xsd:Name">Mary</name>
<gender xsi:type="ns:Gender">female</gender>
<address xsi:type="ns:Address">
<street href="#3"/>
<number xsi:type="unsignedInt">5</number>
<city href="#4"/>
</address>
</mother>
<father xsi:type="ns:Person">
<name xsi:type="xsd:Name">Stuart</name>
<gender xsi:type="ns:Gender">male</gender>
<address xsi:type="ns:Address">
<street xsi:type="xsd:string">Main st.</street>
<number xsi:type="unsignedInt">13</number>
<city href="#4"/>
</address>
</father>
</johnnie>
```

The following program fragment decodes this content and reconstructs the orignal data structure on the heap:

```
#include "soapH.h"
int main()
{
   struct soap soap;
   ns__Person *mother, *father, *john = NULL;
   soap_init(&soap);
   soap_begin(&soap);
   soap_get_ns__Person(&soap, john, "johnnie", NULL);
   mother = john->mother;
   father = john->father;
   ...
   soap_free(&soap); // Clean up temporary data but keep deserialized data
}
struct Namespace namespaces[] =
{
   {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
   {"SOAP-ENC","http://schemas.xmlsoap.org/soap/encoding/"},
   {"xsi", "http://www.w3.org/1999/XMLSchema-instance"},
   {"xsd", "http://www.w3.org/1999/XMLSchema"},
   {"ns", "urn:person"}, // Namespace URI of the "Person" data type
   {NULL, NULL}
};
```

It is REQUIRED to either pass NULL to the soap_get routine, or a valid pointer to a data structure that can hold the decoded content. The following example explicitly passes NULL:

```
   john = soap_get_ns__Person(&soap, NULL, "johnnie", NULL);
```

Note: the second NULL parameter indicates that the schema type attribute of the receiving message can be ignored. The deserializer stores the SOAP contents on the heap, and returns the address. The allocated storage is released with the soap_end call, which removes all temporary and deserialized data from the heap, or with the soap_free call, which removes all temporary data only.

Alternatively, the SOAP content can be decoded within an existing allocated data structure. The following program fragment decodes the SOAP content in a **struct** ns__Person allocated on the stack:

```
#include "soapH.h"
main()
{
   struct soap soap;
   ns__Person *mother, *father, john;
   soap_init(&soap);
   soap_begin(&soap);
   soap_default_ns__Person(&soap, &john);
   soap_get_ns__Person(&soap, &john, "johnnie", NULL);
   mother = john->mother;
   father = john->father;
   ...
   soap_free(&soap);
}
struct Namespace namespaces[] =
   ...
```

Note the use of soap_default_ns__Person. This routine is generated by the gSOAP stub and skeleton compiler and assigns default values to the fields of john.

5.4.4 Default Values for Deserializing Omitted Data

The gSOAP compiler generates soap_default functions for all data types. The default values of the primitive types can be easily changed by defining any of the following macros in the stdsoap2.h file:

```
#define SOAP_DEFAULT_bool
#define SOAP_DEFAULT_byte
#define SOAP_DEFAULT_double
#define SOAP_DEFAULT_float
#define SOAP_DEFAULT_int
#define SOAP_DEFAULT_long
#define SOAP_DEFAULT_LONG64
#define SOAP_DEFAULT_short
#define SOAP_DEFAULT_string
#define SOAP_DEFAULT_time
#define SOAP_DEFAULT_unsignedByte
#define SOAP_DEFAULT_unsignedInt
#define SOAP_DEFAULT_unsignedLong
#define SOAP_DEFAULT_unsignedLONG64
#define SOAP_DEFAULT_unsignedShort
#define SOAP_DEFAULT_wstring
```

Instead of adding these to stdsoap2.h, you can also compile with option -DWITH_USERDEFS_H and include your definitions in file userdefs.h. The absence of a data value in a receiving SOAP message will result in the assignment of a default value to a primitive type upon deserialization.

Default values can also be assigned to individual **struct** and **class** fields of primitive type. For example,

```
struct MyRecord
{
  char *name = "Unknown";
  int value = 9999;
  enum Status { active, passive } status = passive;
}
```

These default values are implicitly assigned to the fields when data values are absent in a receiving SOAP message.

6 Using the gSOAP Stub and Skeleton Compiler

The gSOAP stub and skeleton compiler is invoked from the command line and optionally takes the name of a header file as an argument or, when the file name is absent, parses the standard input:

```
soapcpp2 [aheaderfile.h]
```

where aheaderfile.h is a standard C++ header file. The compiler acts as a preprocessor and produces C++ source files that can be used to build SOAP client and Web service applications in C++. The files generated by the compiler are:

File Name	Description
soapH.h	Main header file to be included by all client and service sources
soapC.cpp	Serializers and deserializers for the specified data structures
soapClient.cpp	Client stub routines and proxies for all remote methods
soapServer.cpp	Service skeleton routines
soapStub.h	A modified header file produced from the compiler input header file
.xsd	An ns.xsd file is generated with an XML schema for each namespace prefix ns used by a data structure in the header file input to the compiler, see Section 5.2.5
.wsdl	A ns.wsdl file is generated with an WSDL description for each namespace prefix ns used by a remote method in the header file input to the compiler, see Section 5.2.5
.nsmap	A ns.nsmap file is generated for each namespace prefix ns used by a remote method in the header file input to the compiler, see Section 5.2.5. The file contains a namespace mapping table that can be used in the client/service sources

Both client and service applications are developed from a header file that specifies the remote methods. If client and service applications are developed with the same header file, the applications are guaranteed to be compatible because the stub and skeleton routines use the same serializers and deserializers ot encode and decode the parameters. Note that when client and service applications are developed together, an application developer does not need to know the details of the internal SOAP encoding used by the client and service.

The following files are part of the gSOAP package and are required to build client and service applications:

File Name	Description
stdsoap2.h	Header file of stdsoap2.cpp runtime library
stdsoap2.c	Runtime C library with XML parser and run-time support routines
stdsoap2.cpp	Runtime C++ library identical to stdsoap2.c

6.1 Compiler Options

The compiler supports the following options:

Option	Description
-h	Print a brief usage message
-c	Save files using extension .c instead of .cpp
-m	Generate code that requires array/binary classes to explicitly free malloced array
-d <path>	Save sources in directory specified by <path>
-p <name>	Save sources with file name prefix <name> instead of "soap"

For example

```
soapcpp2 -cd '../projects' -pmy file.h
```

Saves the sources:

```
../projects/myH.h
../projects/myC.c
../projects/myClient.c
../projects/myServer.c
../projects/myStub.h
```

MS Windows users can use the usual "/" for options, for example:

```
soapcpp2 /cd '..\projects' /pmy file.h
```

6.2 Compiling a SOAP C++ Client

After invoking the gSOAP stub and skeleton compiler on a header file description of a service, the client application can be compiled on a Linux machine as follows:

```
g++ -o myclient myclient.cpp stdsoap2.cpp soapC.cpp soapClient.cpp
```

Or on a Unix machine:

```
g++ -o myclient myclient.cpp stdsoap2.cpp soapC.cpp soapClient.cpp -lsocket -lxnet -lm
```

(Depending on your system configuration, the libraries libsocket.a, libxnet.a, libnsl.a or dynamic *.so versions of those libraries are required.)

The myclient.cpp file must include soapH.h and must define a global namespace mapping table. A typical client program layout with namespace mapping table is shown below:

```
// Contents of file "myclient.cpp"
#include "soapH.h";
...
// A remote method invocation:
  soap_call_some_remote_method(...);
...
```

```
struct Namespace namespaces[] =
{   // {"ns-prefix", "ns-name"}
   {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
   {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"},
   {"xsi", "http://www.w3.org/1999/XMLSchema-instance"},
   {"xsd", "http://www.w3.org/1999/XMLSchema"},
   {"ns1", "urn:my-remote-method"},
   {NULL, NULL}
};
...
```

A mapping table is generated by the gSOAP compiler that can be used in the source, see Section 5.2.5.

6.3 Compiling a SOAP C++ Web Service

After invoking the gSOAP stub and skeleton compiler on a header file description of the service, the server application can be compiled on a Linux machine as follows:

```
g++ -o myserver myserver.cpp stdsoap2.cpp soapC.cpp soapServer.cpp
```

Or on a Unix machine:

```
g++ -o myserver myserver.cpp stdsoap2.cpp soapC.cpp soapServer.cpp -lsocket -lxnet -lm
```

(Depending on your system configuration, the libraries libsocket.a, libxnet.a, libnsl.a or dynamic *.so versions of those libraries are required.)

The myserver.cpp file must include soapH.h and must define a global namespace mapping table. A typical service program layout with namespace mapping table is shown below:

```
// Contents of file "myserver.cpp"
#include "soapH.h";
int main()
{
   soap_serve(soap_new());
}
...
// Implementations of the remote methods as C++ functions
...
struct Namespace namespaces[] =
{   // {"ns-prefix", "ns-name"}
   {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
   {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"},
   {"xsi", "http://www.w3.org/1999/XMLSchema-instance"},
   {"xsd", "http://www.w3.org/1999/XMLSchema"},
   {"ns1", "urn:my-remote-method"},
   {NULL, NULL}
};
...
```

When the serive application is compiled as a CGI application, the soap_serve function acts as a service dispatcher. It listens to standard input and invokes the method via a skeleton routine to serve a SOAP client request. After the request is served, the response is encoded in SOAP and send to standard output. The method must be implemented in the server application and the type signature of the method must be identical to the remote method specified in the header file. That is, the function prototype in the header file must be a valid prototype of the method implemented as a C++ function.

6.4 Using gSOAP for Creating Web Services and Clients in C

The gSOAP compiler can be used to create C (instead of C++) Web services and clients. The gSOAP stub and skeleton compiler soapcpp2 generates .cpp files by default. However, these files only use C syntax and data types **if** the header file input to soapcpp2 uses C syntax and data types. Therefore, a C compiler can be used to compile the .cpp files (e.g. by renaming the extensions to .c) to create C Web service and client executables. For example, with symbolic links on Unix/Linux:

```
ln -s soapC.cpp soapC.c
ln -s soapClient.cpp soapClient.c
ln -s soapServer.cpp soapServer.c
soapcpp2 quote.h
gcc quote.c stdsoap2.c soapC.c soapClient.c
```

6.5 Limitations of gSOAP

gSOAP is fully SOAP 1.1 (and partly SOAP 1.2) compliant and supports all SOAP 1.1 RPC features.

From the perspective of the C/C++ language, a few C++ language features are not supported by gSOAP and these features cannot be used in the specification of SOAP remote methods.

The following C++ language constructs cannot be used by the header file input to the SOAP C++ stub and skeleton compiler:

Templates The SOAP C++ stub and skeleton compiler is a preprocessor and cannot predict the template intantations used by the main program, nor can it generate templated code.

Multiple inheritance Single class inheritance is supported. Multiple inheritance cannot be supported due to limitations of the SOAP protocol.

Abstract methods A class must be instantiatable to allow decoding of instances of the class.

Pragmas All pragmas such as #include and #define are not supported. All pragmas are ignored by the compiler. A traditional C++ preprocessor can be used for the interpretation of pragmas. For example, Unix and Linux users can use "`cpp -B`" to expand the header file, e.g. `cpp -B myfile.h | soapcpp2`.

C and C++ programming statements All class methods of a class should be declared within the class declaration in the header file, but the methods should not be implemented in code. All class method implementations must be defined within another C++ source file and linked to the application.

In addition, the following data types cannot be used in the header file (they can, however be used as a class method return type and as class method parameter types of a class declared in the header file):

union types Because the run-time value of a **union** data type cannot be determined by the compiler, the data type cannot be encoded. An alternative is to use a **struct** with a pointer type for each field. Because NULL pointers are not encoded, the resulting encoding will appear as a union type if only one pointer field is valid (i.e. non-NULL) at the time that the data type is encoded.

void and void* types The **void** data type cannot be encoded. The **void*** data type is typically used to point to some object or to some array of some type of objects at run-time. The compiler cannot determine the type of data pointed to and the size of the array pointed to.

Pointers to sequences of elements in memory Any pointer, except for C strings which are pointers to a sequence of characters, are treated by the compiler as if the pointer points to **only one element in memory** at run-time. Consequently, the encoding and decoding routines will ignore any subsequent elements that follow the first in memory. For the same reason, arrays of undetermined length, e.g. **float** a[] cannot be used. gSOAP supports dynamic arrays using a special type convention, see Section 8.8.

Uninitialized pointers Obviously, all pointers that are part of a data structure must be valid or NULL to enable serialization of the data structure at run time.

There are a number of programming solutions that can be adopted to circumvent these limitations. Instead of using **void***, a program can in some cases be modified to use a pointer to a known type. If the pointer is intended to point to different types of objects, a generic base class can be declared and the pointer is declared to point to the base class. All the other types are declared to be derived classes of this base class. For pointers that point to a sequence of elements in memory dynamic arrays should be used instead, see 8.8.

6.6 gSOAP Serialization Options and Flags

The gSOAP runtime environment flag variables can be set (i.e. 1, where default is 0 which means off) to control the SOAP XML serialization of data with gSOAP. Although gSOAP is fully SOAP 1.1 compliant, some SOAP implementations may have trouble accepting multi-reference data and implicit null data so these flags can be used to put gSOAP in "safe mode". In addition, the embedding of multi-reference data is a feature that is likely to be adopted in future SOAP specifications which gSOAP already supports (turned off by default). The flags are:

Flag	Description
soap.disable_href	Do not serialize multi-reference data, but copy data in SOAP payload
soap.enable_embedding	Embed multi-reference data instead of encoding independent elements
soap.enable_null	Always output null pointers with XML `xsi:nil="true"` attribute Fault when `xsi:nil="true"` attribute is received for a non-pointer data type (normally default value)
soap.enable_utf_string	Store and emit UTF8/16 encoded strings (**char***) without translation
soap.disable_request_count	Do not include HTTP Content-Length in service request
soap.disable_response_count	Do not include HTTP Content-Length in service response (use this option for CGI applications as the Web server determines Content-Length)
soap.enable_array_overflow	Do not fault when receiving excess elements that do not fit in a fixed-size array

The flags can also be selectively turned on/off when multiple Web services are accessed by a client. The flags control the serialization only. Deserialization can handle all different serialization formats automatically.

Caution: Disabling hrefs (multi-reference data output) can be used to improve interoperability with SOAP implementations that are not fully SOAP 1.1 compliant. However, disabling hrefs will crash the serializer for cyclic data structures.

6.7 Memory Management

Understanding gSOAP's run-time memory management is important to optimize client and service applications by eliminating memory leaks and/or dangling references.

There are two forms of dynamic (heap) allocations made by gSOAP's runtime for serialization and deserialization of data. Temporary data is created by the runtime such as hash tables to keep pointer reference information for serialization and hash tables to keep XML id/href information for multi-reference object deserialization. Deserialized data is created upon receiving SOAP messages. This data is stored on the heap and involves calls to the malloc library function and **new** to create class instances. All such allocations are tracked by gSOAP's runtime by linked lists for later deallocation. The linked list for malloc allocations uses some extra space in each malloced block to form a chain of pointers through the malloced blocks. A separate malloced linked list is used to keep track of class instance allocations.

gSOAP does not enforce a deallocation policy and the user can adopt a deallocation policy that works best for a particular application. As a consequence, deserialized data is never deallocated by the gSOAP runtime unless the user explicitly forces deallocation by calling deallocation functions.

The deallocation functions are:

Function Call	**Description**
soap_end(**struct** soap *soap)	Remove temporary data and deserialized data except class instances
soap_free(**struct** soap *soap)	Remove temporary data only
soap_destroy(**struct** soap *soap)	Remove all dynamically allocated class instances Need to be called before soap_end()
soap_dealloc(**struct** soap *soap, **void** *p)	Remove data/object at p. When p==NULL: remove all dynamically allocated (deserialized) data except class instances
soap_unlink(**struct** soap *soap, **void** *p)	Unlink data/object at p from gSOAP's deallocation chain so gSOAP won't deallocate it
soap_done(**struct** soap *soap)	Reset: close (master/slave) sockets and remove callbacks (Section 12.6

Temporary data (i.e. the hash tables) are automatically removed with a call to the soap_free function when the next call to a stub or skeleton routine is made. Deallocation of non-class based data is straightforward: soap_end removes all dynamically allocated deserialized data (data allocated with soap_malloc. That is, when the client/service application does not use any class instances that are (de)marshalled, but uses structs, arrays, etc., then calling the soap_end function is safe to remove all deserialized data. The function can be called after processing the deserialized data of a remote method call or after a number of remote method calls have been made. The function is also typically called after soap_serve, when the service finished sending the response to a client and the deserialized client request data can be removed.

There are three situations to consider for memory deallocation policies for class instances:

1. the program code deletes the class instances and the class destructors in turn SHOULD delete and free any dynamically allocated data (deep deallocation) without calling the soap_end and soap_destroy functions,
2. or the class destructors SHOULD NOT deallocate any data and the soap_end and soap_destroy functions can be called to remove the data.
3. or the class destructors SHOULD mark their own deallocation and mark the deallocation of any other data deallocated by it's destructors by calling the soap_unlink function. This allows soap_destroy and soap_end to remove the remaining instances and data without causing duplicate deallocations.

With the -m option of soapcpp2 enabled, there is one exception which requires explicit deallocation of malloced data in the destructors of classes for array binary types:

- A dynamic array class with non-class elements SHOULD delete the contents of the array it points to as part of its destructor's operations (this includes classes for hexBinary and base64Binary schema types.

It is adviced to use pointers to class instances that are used within other structs and classes to avoid the creation of temporary class instances. The problem with temporary class instances is that the destructor of the temporary may affect data used by other instances through the sharing of data parts accessed with pointers. A dynamic array of class instances is similar: temporaries may be created to fill the array upon deserialization. To avoid problems, use dynamic arrays of pointers to class instances. This also enables the exchange of polymorphic arrays when the elements are instances of classes in an inheritance hierarchy. To summarize, it is adviced to pass class data types by pointer to a remote method. For example:

```
class X { ... };
ns__remoteMethod(X *in, ...);
```

Response elements that are class data types can be passed by reference, as in:

```
class X { ... };
class ns__remoteMethodResponse { ... };
ns__remoteMethod(X *in, ns__remoteMethodResponse &out);
```

But dynamic arrays declared as class data types should use a pointer to a valid object that will be overwritten, as in:

```
typedef int xsd__int;
class X { ... };
class ArrayOfint { xsd__int *__ptr; int __size; };
ns__remoteMethod(X *in, ArrayOfint *out);
```

Or a reference to a valid or NULL pointer, as in:

```
typedef int xsd__int;
class X { ... };
class ArrayOfint { xsd__int *__ptr; int __size; };
ns__remoteMethod(X *in, ArrayOfint *&out);
```

The gSOAP memory allocation functions can be used in client and/or service code to allocate temporary data that will be automatically deallocated. These functions are:

Function Call	**Description**
void *soap_malloc(**struct** soap *soap, size_t n)	return pointer to n bytes
Class *soap_new_Class(**struct** soap *soap, **int** n)	instantiate n CLass objects

The soap_new_X functions are generated by the gSOAP compiler for every class X in the header file. Parameter n MUST be -1 to instantiate a single object or ≥ 0 to instantiate an array of n objects.

Space allocated with soap_malloc will be released with the soap_end and soap_dealloc functions. The objects instantiated with soap_new_X are removed with soap_destroy. For example, the following service uses temporary data in the remote method implementation:

```
int main()
{ ...
  struct soap soap;
  soap_init(&soap);
  soap_serve(&soap);
  soap_end(&soap);
  ...
}
```

An example remote method that allocates a temporary string is:

```
int ns__itoa(struct soap *soap, int i, char **a)
{
  *a = (char*)soap_malloc(soap, 11);
  sprintf(*a, "%d", i);
  return SOAP_OK;
}
```

This temporary allocation can also be used to allocate strings for the SOAP Fault data structure. For example:

```
int ns__mymethod(...)
{ ...
  if (exception)
  {
    soap_fault(soap); // allocate Fault data
    soap->fault->faultstring = (char*)soap_malloc(soap, 1024);
    strcpy(soap->fault->faultstring, ...);
    return SOAP_FAULT;
  }
  ...
}
```

When a class has a **struct** soap * field, this field will be set to point to the current gSOAP run-time environment by gSOAP's deserializers. This simplifies memory management for class instances. The **struct** soap* pointer is implicitly set by the gSOAP deserializer for the class or explicitly by calling the soap_new_X function for class X. For example:

```
class Sample
{ public:
   struct soap *soap; // reference to gSOAP's run-time
   ...
   Sample();
   ~Sample();
};
```

The constructor and destructor for class Sample are:

```
Sample::Sample()
{ this->soap = NULL;
}
Sample::~Sample()
{ soap_unlink(this->soap, this);
}
```

The soap_unlink() call removes the object from gSOAP's deallocation chain. In that way, soap_destroy can be safely called to remove all class instances. The following code illustrates the explicit creation of a Sample object and cleanup:

```
struct soap *soap = soap_new(); // new gSOAP runtime
Sample *obj = soap_new_Sample(soap, -1); // new Sample object with obj->soap set to runtime
...
delete obj; // also calls soap_unlink to remove obj from the deallocation chain
soap_destroy(soap); // deallocate all (other) class instances
soap_end(soap); // clean up
```

Here is another example:

```
class ns__myClass
{ ...
   struct soap *soap; // set by soap_new_ns__myClass()
   char *name;
   void setName(const char *s);
   ...
};
```

Calls to soap_new_ns__myClass(soap, n) will set the soap field in the class instance to the current gSOAP environment. Because the deserializers invoke the soap_new functions, the soap field of the ns__myClass instances are set as well. This mechanism is convenient when Web Service methods need to return objects that are instantiated in the methods. For example

```
int ns__myMethod(struct soap *soap, ...)
{
   ns__myClass *p = soap_new_ns__myClass(soap, -1);
   p-¿setName("SOAP");
   return SOAP_OK;
}
void ns__myClass::ns__setName(const char *s)
{
   if (soap)
      name = (char*)soap_malloc(soap, strlen(s)+1);
   else
      name = (char*)malloc(strlen(s)+1);
   strcpy(name, s);
}
ns__myClass::ns__myClass()
{
   soap = NULL;
   name = NULL;
}
ns__myClass::~ns__myClass()
{
   if (!soap && name) free(name);
   soap_unlink(soap, this);
}
```

Calling soap_destroy right after soap_serve in the Web Service will destroy all dynamically allocated class instances.

6.8 Debugging

To activate message logging for debugging, un-comment the #define DEBUG pragma in stdsoap2.h. Compile the client and/or server applications as described above (or simply use g++ -DDEBUG ... to compile with debugging activated). When the client and server applications run, they will log their activity in three separate files:

File	Description
SENT.log	The SOAP content transmitted by the application
RECV.log	The SOAP content received by the application
TEST.log	A log containing various activities performed by the application

Caution: The client and server applications may run slow due to the logging activity.

Caution: When installing a CGI application on the Web with debugging activated, the log files may sometimes not be created due to file access permission restrictions imposed on CGI applications. To get around this, create empty log files with universal write permissions. Be careful about the security implication of this.

You can test a service CGI application without deploying it on the Web. To do this, create a client application for the service and activate message logging by this client. Remove any old SENT.log file and run the client (which connects to the Web service or to another dummy, but valid address) and copy the SENT.log file to another file, e.g. SENT.tst. Then redirect the SENT.tst file to the service CGI application. For example,

```
myservice.cgi < SENT.tst
```

This should display the service response on the terminal.

The file names of the log files and the logging activity can be controlled at the application level. This allows the creation of separate log files by separate services, clients, and threads. For example, the following service logs all SOAP messages (but no debug messages) in separate directories:

```
struct soap soap;
soap_init(&soap);
...
soap_set_recv_logfile(&soap, "logs/recv/service12.log"); // append all messages received in /logs/recv/service12.log
soap_set_sent_logfile(&soap, "logs/sent/service12.log"); // append all messages sent in /logs/sent/service12.log
soap_set_test_logfile(&soap, NULL); // no file name: do not save debug messages
...
soap_serve(&soap);
...
```

Likewise, messages can be logged for individual client-side remote method calls.

6.9 Libraries

- The socket library is essential and requires the inclusion of the appropriate libraries with the compile command for Sun Solaris systems:

  ```
  g++ -o myclient myclient.cpp stdsoap2.cpp soapC.cpp soapClient.cpp -lsocket -lxnet -lnsl -lm
  ```

  These library loading options are not required with Linux.

- The gSOAP runtime uses the math library for the `NaN`, `INF`, and `-INF` floating point representations. The library is not strictly necessary and the ¡math.h¿ header file import can be commented out from the stdsoap2.h header file. Then, the application can be linked without the -lm math library under Sun Solaris:

  ```
  g++ -o myclient myclient.cpp stdsoap2.cpp soapC.cpp soapClient.cpp -lsocket -lxnet -lnsl
  ```

7 The gSOAP Remote Method Specification Format

A SOAP remote method is specified as a C/C++ function prototype in a header file. The function is REQUIRED to return **int**, which is used to represent a SOAP error code, see Section 7.2. Multiple remote methods MAY be declared together in one header file.

The general form of a SOAP remote method specification is:

```
[int] [namespace_prefix__]method_name([inparam1, inparam2, ...,] outparam);
```

where

namespace_prefix__ is the optional namespace prefix of the method (see identifier translation rules 7.3)

method_name it the remote method name (see identifier translation rules 7.3)

inparam is the declaration of an input parameter of the remote method

outparam is the declaration of the output parameter of the remote method

This simple form can only pass a single, non-**struct** and non-**class** type output parameter. See 7.1 for passing multiple output parameters. The name of the declared function namespace_prefix__method_name must be unique and cannot match the name of a **struct**, **class**, or **enum** declared in the same header file.

The method request is encoded in SOAP as an XML element and the namespace prefix, method name, and input parameters are encoded using the format:

```
<[namespace-prefix:]method_name xsi:type="[namespace-prefix:]method_name>
<inparam-name1 xsi:type="...">...</inparam-name1>
<inparam-name2 xsi:type="...">...</inparam-name2>
...
</[namespace-prefix:]method_name>
```

where the `inparam-name` accessors are the element-name representations of the inparam parameter name declarations, see Section 7.3. (The optional parts are shown enclosed in [].)

The XML response by the Web service is of the form:

```
<[namespace-prefix:]method-nameResponse xsi:type="[namespace-prefix:]method-nameResponse>
<outparam-name xsi:type="...">...</outparam-name>
</[namespace-prefix:]method-nameResponse>
```

where the `outparam-name` accessor is the element-name representation of the outparam parameter name declaration, see Section 7.3. By convention, the response element name is the method name ending in `Response`. See 7.1 on how to change the declaration if the service response element name is different.

The gSOAP stub and skeleton compiler generates a stub routine and a proxy for the remote method. This proxy is of the form:

```
int soap_call_[namespace_prefix__]method_name(struct soap *soap, char *URL, char *action, [inparam1,
inparam2, ...,] outparam);
```

This proxy can be called by a client application to perform the remote method call.

The gSOAP stub and skeleton compiler generates a skeleton routine for the remote method. The skeleton function is:

```
int soap_serve_[namespace_prefix__]method_name(struct soap *soap);
```

The skeleton routine, when called by a service application, will attempt to serve a request on the standard input. If no request is present or if the request does not match the method name, SOAP_NO_METHOD is returned. The skeleton routines are automatically called by the generated soap_serve routine that handles all requests.

7.1 Remote Method Parameter Passing

The input parameters of a remote method MUST be passed by value. Input parameters cannot be passed by reference with the & reference operator, but an input parameter value MAY be passed using a pointer to the data. Passing a pointer to the data is prefered when the size of the data of the parameter is large. Also, to pass instances of (derived) classes, pointers to the instance need to be used to avoid passing the instance by value which requires a temporary and prohibits passing derived class instances. When two input parameter values are identical, passing them using a pointer has the advantage that the value will be encoded only once as multi-reference (hence, the parameters are aliases). When input parameters are passed using a pointer, the data pointed to will not be modified by the remote method and returned to the caller.

The output parameter MUST be passed by reference using & or by using a pointer. Arrays are passed by reference by default and do not require the use of the reference operator &.

The input and output parameter types have certain limitations, see Section 6.5

If the output parameter is a **struct** or **class** type, it is considered a SOAP remote method response element instead of a simple output parameter value. That is, the name of the **struct** or **class** is the name of the response element and the **struct** or **class** fields are the output parameters of the remote method, see also 5.1.6. Hence, if the output parameter has to be a **struct** or **class**, a response **struct** or **class** MUST be declared as well. In addition, if a remote method returns multiple output parameters, a response **struct** or **class** MUST be declared. By convention, the response element is the remote method name ending with "`Response`".

The general form of a response element declaration is:

```
struct [namespace_prefix__]response_element_name
{
  outparam1;
  outparam2;
  ...
};
```

where

namespace_prefix__ is the optional namespace prefix of the response element (see identifier translation rules 7.3)

response_element_name it the name of the response element (see identifier translation rules 7.3)

outparam is the declaration of an output parameter of the remote method

The general form of a remote method specification with a response element declaration for (multiple) output parameters is:

```
[int] [namespace_prefix__]method_name([inparam1, inparam2, ...,] struct [namespace_prefix__]response_element_name
{outparam1[, outparam2, ...]} &anyparam);
```

The choice of name for anyparam has no effect on the SOAP encoding and decoding and is only used as a place holder for the response.

The method request is encoded in SOAP as an independent element and the namespace prefix, method name, and input parameters are encoded using the format:

```
<[namespace-prefix:]method-name xsi:type="[namespace-prefix:]method-name>
<inparam-name1 xsi:type="...">...</inparam-name1>
<inparam-name2 xsi:type="...">...</inparam-name2>
...
</[namespace-prefix:]method-name>
```

where the inparam-name accessors are the element-name representations of the inparam parameter name declarations, see Section 7.3. (The optional parts resulting from the specification are shown enclosed in [].)

The method response is expected to be of the form:

```
<[namespace-prefix:]response-element-name xsi:type="[namespace-prefix:]response-element-name>
<outparam-name1 xsi:type="...">...</outparam-name1>
<outparam-name2 xsi:type="...">...</outparam-name2>
...
</[namespace-prefix:]response-element-name>
```

where the outparam-name accessors are the element-name representations of the outparam parameter name declarations, see Section 7.3. (The optional parts resulting from the specification are shown enclosed in [].)

The input and/or output parameters can be made anonymous, which allows the deserialization of requests/responses with different parameter names as is endorsed by the SOAP 1.1 specification, see Section 5.1.12.

7.2 Stub and Skeleton Routine Error Codes

The error codes returned by the stub and skeleton routines are listed below.

#	Code	Description
0	SOAP_OK	No error
1	SOAP_CLI_FAULT*	The service raised a client fault exception
2	SOAP_SVR_FAULT*	The service raised a server fault exception
3	SOAP_TAG_MISMATCH	An XML element didn't correspond to anything expected
4	SOAP_TYPE_MISMATCH	An XML schema type mismatch
5	SOAP_SYNTAX_ERROR	An XML syntax error occurred on the input
6	SOAP_NO_TAG	Begin of an element expected, but not found
7	SOAP_IOB	Array index out of bounds
8	SOAP_MUSTUNDERSTAND*	An element needs to be ignored that need to be understood
9	SOAP_NAMESPACE	Namespace name mismatch (validation error)
10	SOAP_OBJ_MISMATCH	Mismatch in the size and/or shape of an object
11	SOAP_FATAL_ERROR	Internal error
12	SOAP_FAULT	An exception raised by the service
13	SOAP_NO_METHOD	Skeleton error: the skeleton cannot serve the method
14	SOAP_EOM	Out of memory
15	SOAP_NULL	An element was null, while it is not supposed to be null
16	SOAP_MULTI_ID	Multiple occurrences of the same element ID on the input
17	SOAP_MISSING_ID	Element ID missing for an HREF on the input
18	SOAP_HREF	Reference to object is incompatible with the object refered to
19	SOAP_TCP_ERROR	A TCP connection error occured
20	SOAP_HTTP_ERROR	An HTTP error occured
21	SOAP_SSL_ERROR	An SSL error occured
22	SOAP_DIME_ERROR	DIME parsing error
23	SOAP_EOD	End of DIME
24	SOAP_VERSIONMISMATCH*	SOAP version mismatch or no SOAP message
25	SOAP_DIME_VERSIONMISMATCH*	DIME version mismatch
-1	SOAP_EOF	Unexpected end of file or no input

The error codes that are returned by a stub routine (proxy) upon receiving a SOAP Fault from the server are marked (*). The remaining error codes are generated by the proxy itself as a result of problems with a SOAP payload. The error code is SOAP_OK when the remote method call was successful (the SOAP_OK predefined constant is guaranteed to be 0). The error code is also stored in soap.error, where soap is a variable that contains the current runtime environment. The function soap_print_fault(**struct** soap *soap, FILE *fd) can be called to display an error message on fd where current value of the soap.error variable is used by the function to display the error. The function soap_print_fault_location(**struct** soap *soap, FILE *fd) prints the location of the error if the error is a result from parsing XML.

A remote method implemented in a SOAP service MUST return an error code as the function's return value. SOAP_OK denotes success and SOAP_FAULT denotes an exception. The exception details can be assigned to the strings soap.fault->faultstring and soap.fault->detail, where soap is a variable that contains the current runtime environment, see Section 9.

7.3 C++ Identifier Name to XML Element Name Translation

One of the secrets behind the power and flexibility of gSOAP's encoding and decoding of remote method names, class names, type identifiers, and struct or class fields is the ability to specify namespace prefixes with these names that are used to denote their encoding style. More specifically, a C/C++ identifier name of the form

```
[namespace_prefix__]element_name
```

will be encoded in XML as

```
<[namespace-prefix:]element-name ...>
```

The **underscore pair** (__) separates the namespace prefix from the element name. Each namespace prefix has a namespace URI specified by a namespace mapping table 7.4, see also Section 5.1.2. The namespace URI is a unique identification that can be associated with the remote methods and data types. The namespace URI disambiguates potentially identical remote method names and data type names used by disparate organizations.

XML element names are NCNames (restricted strings) that MAY contain **hypens**, **dots**, and **underscores**. The special characters in the XML element names of remote methods, structs, classes, typedefs, and fields can be controlled using the following conventions: A **single underscore** in a namespace prefix or identifier name is replaced by a hyphen (-) in the XML element name. For example, the identifier name SOAP_ENC__ur_type is represented in XML as SOAP-ENC:ur-type. The sequence _DOT_ is replaced by a dot (.), and the sequence _USCORE_ is replaced by an underscore (_) in the corresponding XML element name. For example:

```
class n_s__biz_DOT_com
{
  char *n_s__biz_USCORE_name;
};
```

is encoded in XML as:

```
<n-s:biz.com xsi:type="n-s:biz.com">
  <n-s:biz_name xsi:type="string">Bizybiz</n-s:biz_name>
</n-s:biz.com>
```

Trailing underscores of an identifier name are not translated into the XML representation. This is useful when an identifier name clashes with a C++ keyword. For example, `return` is often used as an accessor name in a SOAP response element. The `return` element can be specified as `return_` in the C++ source code. Note that XML should be treated as case sensitive, so the use of e.g. `Return` may not always work to avoid a name clash with the **return** keyword. The use of trailing underscores also allows for defining **struct**s and **class**es with essentially the same XML schema type name, but that have to be distinghuished as seperate C/C++ types.

For decoding, the underscores in identifier names act as wildcards. An XML element is parsed and matches the name of an identifier if the name is identical to the element name (case insensitive) and the underscores in the identifier name are allowed to match any character in the element name. For example, the identifier name `I_want__soap_fun_the_bea___DOT_com` matches the element name `I-want:SOAP4fun@the-beach.com`.

7.4 Namespace Mapping Table

A namespace mapping table MUST be defined by clients and service applications. The mapping table is used by the serializers and deserializers of the stub and skeleton routines to produce a valid SOAP payload and to validate an incoming SOAP payload. A typical mapping table is shown below:

```
struct Namespace namespaces[] =
{  // {"ns-prefix", "ns-name"}
  {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"}, // MUST be first
  {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"}, // MUST be second
  {"xsi", "http://www.w3.org/1999/XMLSchema-instance"}, // MUST be third
  {"xsd", "http://www.w3.org/1999/XMLSchema"}, // Required for XML schema types
  {"ns1", "urn:my-service-URI"}, // The namespace URI of the remote methods
  {NULL, NULL} // end of table
};
```

Each namespace prefix used by a identifier name in the header file specification (see Section 7.3) MUST have a binding to a namespace URI in the mapping table. The end of the namespace mapping table MUST be indicated by the `NULL` pair. The namespace URI matching is case insensitive. A namespace prefix is distinghuished by the occurrence of a pair of underscores (__) in an identifier.

An optional namespace pattern MAY be provided with each namespace mapping table entry. The patterns provide an alternative namespace matching for the validation of decoded SOAP messages. In this pattern, dashes (-) are single-character wildcards and asterisks (*) are multi-character wildcards. For example, to decode different versions of XML Schema type with different authoring dates, four dashes can be used in place of the specific dates in the namespace mapping table pattern:

```
struct Namespace namespaces[] =
{  // {"ns-prefix", "ns-name", "ns-name validation pattern"}
```

```
...
   {"xsi", "http://www.w3.org/1999/XMLSchema-instance", "http://www.w3.org/----/XMLSchema-
instance"},
   {"xsd", "http://www.w3.org/1999/XMLSchema", "http://www.w3.org/----/XMLSchema"},
...
```

Or alternatively, asterisks can be used as wildcards for multiple characters:

```
struct Namespace namespaces[] =
{   // {"ns-prefix", "ns-name", "ns-name validation pattern"}
...
   {"xsi", "http://www.w3.org/1999/XMLSchema-instance", "http://www.w3.org/*/XMLSchema-
instance"},
   {"xsd", "http://www.w3.org/1999/XMLSchema", "http://www.w3.org/*/XMLSchema"},
...
```

A namespace mapping table is automatically generated together with a WSDL file for each namespace prefix that is used for a remote method in the header file. This namespace mapping table has entries for all namespace prefixes. The namespace URIs need to be filled in. These appear as %{URI}% in the table. See Section 12.2 on how to specify the namespace URIs in the header file.

For decoding elements with namespace prefixes, the namespace URI associated with the namespace prefix (through the `xmlns` attribute of an XML element) is searched from the beginning to the end in a namespace mapping table, and for every row the following tests are performed as part of the validation process:

1. the string in the second column matches the namespace URI (case insensitive)
2. the string in the optional third column matches the namespace URI (case insensitive), where - is a one-character wildcard and * is a multi-character wildcard

When a match is found, the namespace prefix in the first column of the table is considered semantically identical to the namespace prefix used by the XML element to be decoded, though the prefix names may differ. A service will respond with the namespace that it received from a client in case it matches a pattern in the third column.

For example, let's say we have the following structs:

```
struct a__elt { ... };
struct b__elt { ... };
struct k__elt { ... };
```

and a namespace mapping table in the program:

```
struct Namespace namespaces[] =
{   // {"ns-prefix", "ns-name", "ns-name validation pattern"}
...
   {"a", "some uri"},
   {"b", "other uri"},
   {"c", "his uri", "* uri"},
...
```

Then, the following XML elements will match the structs:

```
<n:elt xmlns:n="some URI">     matches the struct name a__elt
...
<m:elt xmlns:m="other URI">     matches the struct name b__elt
...
<k:elt xmlns:k="my URI">     matches the struct name c__elt
...
```

The response of a service to a client request that uses the namespaces listed above, will include `my URI` for the name space of element `k`.

It is possible to use a number of different namespace tables and select the one that is appropriate. For example, an application might contact many different Web services all using different namespace URIs. If all the URIs are stored in one table, each remote method invocation will dump the whole namespace table in the SOAP payload. There is no technical problem with that, but it can be ugly when the table is large. To use different namespace tables, declare a pointer to a table and set the pointer to a particular table before remote method invocation. For example:

```
struct Namespace namespacesTable1[] = { ... };
struct Namespace namespacesTable2[] = { ... };
struct Namespace namespacesTable3[] = { ... };
struct Namespace *namespaces;
...
struct soap soap;
...
soap_init(&soap);
soap.namespaces = namespaceTable1;
soap_call_remote_method(&soap, URL, Action, ...);
...
```

8 gSOAP Serialization and Deserialization Rules

This section describes the serialization and deserialization of C and C++ data types for SOAP 1.1 and 1.2 compliant encoding and decoding.

8.1 Primitive Type Encoding

The default encoding rules for the primitive C and C++ data types are given in the table below:

Type	Schema Type	Example Encoding
bool	`boolean`	`<boolean xsi:type="boolean">...</boolean>`
char* (C string)	`string`	`<string xsi:type="string">...</string>`
char	`byte`	`<byte xsi:type="byte">...</byte>`
double	`double`	`<double xsi:type="double">...</double>`
float	`float`	`<float xsi:type="float">...</float>`
int	`int`	`<int xsi:type="int">...</int>`
long	`long`	`<long xsi:type="long">...</long>`
LONG64	`long`	`<long xsi:type="long">...</long>`
long long	`long`	`<long xsi:type="long">...</long>`
short	`short`	`<short xsi:type="short">...</short>`
time_t	`dateTime`	`<dateTime xsi:type="dateTime">...</dateTime>`
unsigned char	`unsignedByte`	`<unsignedByte xsi:type="unsignedByte">...</unsignedByte>`
unsigned int	`unsignedInt`	`<unsignedInt xsi:type="unsignedInt">...</unsignedInt>`
unsigned long	`unsignedLong`	`<unsignedLong xsi:type="unsignedLong">...</unsignedLong>`
ULONG64	`unsignedLong`	`<unsignedLong xsi:type="unsignedLong">...</unsignedLong>`
unsigned long long	`unsignedLong`	`<unsignedLong xsi:type="unsignedLong">...</unsignedLong>`
unsigned short	`unsignedShort`	`<unsignedShort xsi:type="unsignedShort">...</unsignedShort>`
wchar_t*	`string`	`<string xsi:type="string">...</string>`

Objects of type **void** and **void*** cannot be encoded.

8.2 How to Encode and Decode Primitive Types as Built-In XML Schema Types

By default, encoding of the primitive types will take place as per SOAP encoding style. The encoding can be changed to any XML schema type with an optional namespace prefix by using a **typedef** in the header file input to the gSOAP stub and skeleton compiler. The declaration enables the implementation of built-in XML schema types such as `positiveInteger`, `xsd:anyURI`, and `xsd:date` for which no built-in data structures in C and C++ exist but which can be represented using standard data structures such as strings, integers, and floats.

The **typedef** declaration is frequently used for convenience in C. A **typedef** declares a type name for a (complex) type expression. The type name can then be used in other declarations in place of the more complex type expression, which often improves the readability of the program code.

The gSOAP compiler interprets **typedef** declarations the same way as a regular C compiler interprets them, i.e. as types in declarations. In addition however, the gSOAP compiler will also use the type name in the encoding of the data in SOAP. The **typedef** name will appear as the XML element name of an independent element and as the value of the `xsi:type` attribute in the SOAP payload.

Many built-in primitive and derived XML schema types such as `xsd:anyURI`, `positiveInteger`, and `decimal` can be stored by standard primitive data structures in C++ as well such as strings, integers, floats, and doubles. To serialize strings, integers, floats, and doubles as built-in primitive and derived XML schema types. To this end, a **typedef** declaration can be used to declare an XML Schema type.

For example, the declaration

```
typedef unsigned int positiveInteger;
```

creates a named type positiveInteger which is represented by **unsigned int** in C++. For example, the encoding of a positiveInteger value 3 is

```
<positiveInteger xsi:type="positiveInteger">3</positiveInteger>
```

The built-in primitive and derived numerical XML Schema types are listed below together with their recommended **typedef** declarations. Note that the SOAP encoding schemas for primitive types are derived from the built-in XML schema types, so SOAP_ENC__ can be used as a namespace prefix instead of xsd__.

`xsd:anyURI` Represents a Uniform Resource Identifier Reference (URI). Each URI scheme imposes specialized syntax rules for URIs in that scheme, including restrictions on the syntax of allowed fragement identifiers. It is recommended to use strings to store `xsd:anyURI` XML schema types. The recommended type declaration is:

```
typedef char *xsd__anyURI;
```

`xsd:base64Binary` Represents Base64-encoded arbitrary binary data. For using the `xsd:base64Binary` XML schema type, the use of the base64Binary representation of a dynamic array is **strongly** recommended, see Section 8.9. However, the type can also be declared as a string and the encoding will be string-based:

```
typedef char *xsd__base64Binary;
```

With this approach, it is solely the responsibility of the application to make sure the string content is according to the Base64 Content-Transfer-Encoding defined in Section 6.8 of RFC 2045.

`xsd:boolean` For declaring an `xsd:boolean` XML schema type, the use of a bool is **strongly** recommended. If a pure C compiler is used that does not support the bool type, see Section 8.3.5. The corresponding type declaration is:

```
typedef bool xsd__boolean;
```

Type xsd__boolean declares a Boolean (0 or 1), which is encoded as

```
<xsd:boolean xsi:type="xsd:boolean">...</xsd:boolean>
```

`xsd:byte` Represents a byte (-128...127). The corresponding type declaration is:

```
typedef char xsd__byte;
```

Type xsd__byte declares a byte which is encoded as

```
<xsd:byte xsi:type="xsd:byte">...</xsd:byte>
```

`xsd:dateTime` Represents a date and time. The lexical representation is according to the ISO 8601 extended format CCYY-MM-DDThh:mm:ss where "CC" represents the century, "YY" the year, "MM" the month and "DD" the day, preceded by an optional leading "-" sign to indicate a negative number. If the sign is omitted, "+" is assumed. The letter "T" is the date/time separator and "hh", "mm", "ss" represent hour, minute and second respectively. It is recommended to use the time_t type to store `xsd:dateTime` XML schema types and the type declaration is:

```
typedef time_t xsd__dateTime;
```

However, note that calendar times before the year 1902 or after the year 2037 cannot be represented. Upon receiving a date below this range, the time_t value will be set to -2147483648, and upon receiving a date above this range, the time_t value will be set to 2147483647.

Strings (**char***) can be used to store `xsd:dateTime` XML schema types. The type declaration is:

```
typedef char *xsd__dateTime;
```

In this case, it is up to the application to read and set the dateTime representation.

`xsd:date` Represents a date. The lexical representation for date is the reduced (right truncated) lexical representation for dateTime: CCYY-MM-DD. It is recommended to use strings (**char***) to store `xsd:date` XML schema types. The type declaration is:

```
typedef char *xsd__date;
```

`xsd:decimal` Represents arbitrary precision decimal numbers. It is recommended to use the **double** type to store `xsd:decimal` XML schema types and the type declaration is:

```
typedef double xsd__decimal;
```

Type xsd__decimal declares a double floating point number which is encoded as

```
<xsd:double xsi:type="xsd:decimal">...</xsd:double>
```

`xsd:double` Corresponds to the IEEE double-precision 64-bit floating point type. The type declaration is:

```
typedef double xsd__double;
```

Type xsd__double declares a double floating point number which is encoded as

```
<xsd:double xsi:type="xsd:double">...</xsd:double>
```

`xsd:duration` Represents a duration of time. The lexical representation for duration is the ISO 8601 extended format PnYn MnDTnH nMnS, where nY represents the number of years, nM the number of months, nD the number of days, T is the date/time separator, nH the number of hours, nM the number of minutes and nS the number of seconds. The number of seconds can include decimal digits to arbitrary precision. It is recommended to use strings (**char***) to store `xsd:duration` XML schema types. The type declaration is:

```
typedef char *xsd__duration;
```

`xsd:float` Corresponds to the IEEE single-precision 32-bit floating point type. The type declaration is:

```
typedef float xsd__float;
```

Type xsd__float declares a floating point number which is encoded as

```
<xsd:float xsi:type="xsd:float">...</xsd:float>
```

`xsd:hexBinary` Represents arbitrary hex-encoded binary data. It has a lexical representation where each binary octet is encoded as a character tuple, consisting of two hexadecimal digits ([0-9a-fA-F]) representing the octet code. For example, "0FB7" is a hex encoding for the 16-bit integer 4023 (whose binary representation is 111110110111. For using the `xsd:hexBinary` XML schema type, the use of the hexBinary representation of a dynamic array is **strongly** recommended, see Section 8.10. However, the type can also be declared as a string and the encoding will be string-based:

```
typedef char *xsd__hexBinary;
```

With this approach, it is solely the responsibility of the application to make sure the string content consists of a sequence of octets.

`xsd:int` Corresponds to a 32-bit integer in the range -2147483648 to 2147483647. If the C++ compiler supports 32-bit **int** types, the type declaration can use the **int** type:

```
typedef int xsd__int;
```

Otherwise, the C++ compiler supports 16-bit **int** types and the type declaration should use the **long** type:

```
typedef long xsd__int;
```

Type xsd__int declares a 32-bit integer which is encoded as

```
<xsd:int xsi:type="xsd:int">...</xsd:int>
```

`xsd:integer` Corresponds to an unbounded integer. Since C++ does not support unbounded integers as a standard feature, the recommended type declaration is:

```
typedef long long xsd__integer;
```

Type xsd__integer declares a 64-bit integer which is encoded as an unbounded `xsd:integer`:

```
<xsd:integer xsi:type="xsd:integer">...</xsd:integer>
```

Another possibility is to use strings to represent unbounded integers and do the translation in code.

`xsd:long` Corresponds to a 64-bit integer in the range -9223372036854775808 to 9223372036854775807. The type declaration is:

```
typedef long long xsd__long;
```

Or in Visual C++:

```
typedef LONG64 xsd__long;
```

Type xsd__long declares a 64-bit integer which is encoded as

```
<xsd:long xsi:type="xsd:long">...</xsd:long>
```

`xsd:negativeInteger` Corresponds to a negative unbounded integer (< 0). Since C++ does not support unbounded integers as a standard feature, the recommended type declaration is:

```
typedef long long xsd__negativeInteger;
```

Type `xsd__negativeInteger` declares a 64-bit integer which is encoded as a `xsd:negativeInteger`:

```
<xsd:negativeInteger xsi:type="xsd:negativeInteger">...</xsd:negativeInteger>
```

Another possibility is to use strings to represent unbounded integers and do the translation in code.

`xsd:nonNegativeInteger` Corresponds to a non-negative unbounded integer (> 0). Since C++ does not support unbounded integers as a standard feature, the recommended type declaration is:

typedef unsigned long long xsd__nonNegativeInteger;

Type `xsd__nonNegativeInteger` declares a 64-bit unsigned integer which is encoded as a non-negative unbounded `xsd:nonNegativeInteger`:

```
<xsd:nonNegativeInteger xsi:type="xsd:nonNegativeInteger">...</xsd:nonNegativeInteger>
```

Another possibility is to use strings to represent unbounded integers and do the translation in code.

`xsd:nonPositiveInteger` Corresponds to a non-positive unbounded integer (≤ 0). Since C++ does not support unbounded integers as a standard feature, the recommended type declaration is:

typedef long long xsd__nonPositiveInteger;

Type `xsd__nonPositiveInteger` declares a 64-bit integer which is encoded as a `xsd:nonPositiveInteger`:

```
<xsd:nonPositiveInteger xsi:type="xsd:nonPositiveInteger">...</xsd:nonPositiveInteger>
```

Another possibility is to use strings to represent unbounded integers and do the translation in code.

`xsd:normalizedString` Represents normalized character strings. Normalized character strings do not contain the carriage return (#xD), line feed (#xA) nor tab (#x9) characters. It is recommended to use strings to store `xsd:normalizeString` XML schema types. The type declaration is:

typedef char *xsd__normalizedString;

Type `xsd__normalizedString` declares a string type which is encoded as

```
<xsd:normalizedString xsi:type="xsd:normalizedString">...</xsd:normalizedString>
```

It is solely the responsibility of the application to make sure the strings do not contain carriage return (#xD), line feed (#xA) and tab (#x9) characters.

`xsd:positiveInteger` Corresponds to a positive unbounded integer (≥ 0). Since C++ does not support unbounded integers as a standard feature, the recommended type declaration is:

typedef unsigned long long xsd__positiveInteger;

Type `xsd__positiveInteger` declares a 64-bit unsigned integer which is encoded as a `xsd:positiveInteger`:

```
<xsd:positiveInteger xsi:type="xsd:positiveInteger">...</xsd:positiveInteger>
```

Another possibility is to use strings to represent unbounded integers and do the translation in code.

`xsd:short` Corresponds to a 16-bit integer in the range -323768 to 323767. The type declaration is:

```
typedef short xsd__short;
```

Type `xsd__short` declares a short 16-bit integer which is encoded as

```
<xsd:short xsi:type="xsd:short">...</xsd:short>
```

`xsd:string` Represents character strings. The type declaration is:

```
typedef char *xsd__string;
```

Type `xsd__string` declares a string type which is encoded as

```
<xsd:string xsi:type="xsd:string">...</xsd:string>
```

The type declaration for wide character strings is:

```
typedef wchar_t *xsd__string;
```

Both type of strings can be used at the same time, but requires one typedef name to be changed by appending an underscore which is invisible in XML. For example:

```
typedef wchar_t *xsd__string_;
```

`xsd:time` Represents a time. The lexical representation for time is the left truncated lexical representation for dateTime: hh:mm:ss.sss with optional following time zone indicator. It is recommended to use strings (**char***) to store `xsd:time` XML schema types. The type declaration is:

```
typedef char *xsd__time;
```

`xsd:token` Represents tokenized strings. Tokens are strings that do not contain the line feed (#xA) nor tab (#x9) characters, that have no leading or trailing spaces (#x20) and that have no internal sequences of two or more spaces. It is recommended to use strings to store `xsd:token` XML schema types. The type declaration is:

```
typedef char *xsd__token;
```

Type `xsd__token` declares a string type which is encoded as

```
<xsd:token xsi:type="xsd:token">...</xsd:token>
```

It is solely the responsibility of the application to make sure the strings do not contain the line feed (#xA) nor tab (#x9) characters, that have no leading or trailing spaces (#x20) and that have no internal sequences of two or more spaces.

`xsd:unsignedByte` Corresponds to an 8-bit unsigned integer in the range 0 to 255. The type declaration is:

```
typedef unsigned char xsd__unsignedByte;
```

Type xsd__unsignedByte declares a unsigned 8-bit integer which is encoded as

```
<xsd:unsignedByte xsi:type="xsd:unsignedByte">...</xsd:unsignedByte>
```

`xsd:unsignedInt` Corresponds to a 32-bit unsigned integer in the range 0 to 4294967295. If the C++ compiler supports 32-bit **int** types, the type declaration can use the **int** type:

```
typedef unsigned int xsd__unsignedInt;
```

Otherwise, the C++ compiler supports 16-bit **int** types and the type declaration should use the **long** type:

```
typedef unsigned long xsd__unsignedInt;
```

Type xsd__unsignedInt declares an unsigned 32-bit integer which is encoded as

```
<xsd:unsignedInt xsi:type="xsd:unsignedInt">...</xsd:unsignedInt>
```

`xsd:unsignedLong` Corresponds to a 64-bit unsigned integer in the range 0 to 18446744073709551615. The type declaration is:

```
typedef unsigned long long xsd__unsignedLong;
```

Or in Visual C++:

```
typedef ULONG64 xsd__unsignedLong;
```

Type xsd__unsignedLong declares an unsigned 64-bit integer which is encoded as

```
<xsd:unsignedLong xsi:type="xsd:unsignedLong">...</xsd:unsignedLong>
```

`xsd:unsignedShort` Corresponds to a 16-bit unsigned integer in the range 0 to 65535. The type declaration is:

```
typedef unsigned short xsd__unsignedShort;
```

Type xsd__unsginedShort declares an unsigned short 16-bit integer which is encoded as

```
<xsd:unsignedShort xsi:type="xsd:unsignedShort">...</xsd:unsignedShort>
```

Other XML schema types such as `gYearMonth`, `gYear`, `gMonthDay`, `gDay`, `xsd:gMonth`, `QName`, `NOTATION`, etc., can be encoded similarly using a **typedef** declaration.

8.2.1 How to Specify Multiple Storage Formats for a Single Primitive XML Schema Type

Trailing underscores (see Section 7.3) can be used in the type name in a **typedef** to enable the declaration of multiple storage formats for a single XML schema type. For example, one part of a C/C++ application's data structure may use plain strings while another part may use wide character strings. To enable this simultaneous use, declare:

```
typedef char *xsd__string;
typedef wchar_t *xsd__string_;
```

Now, the xsd__string and xsd__string_ types will both be encoded and decoded as XML string types and the use of trailing underscores allows multiple declarations for a single XML schema type.

8.2.2 How to Specify Polymorphic Primitive Types

SOAP 1.1 supports polymorphic types, because XML schema types form a hierarchy. The root of the hierarchy is called `xsd:anyType`. So, for example, an array of `xsd:anyType` in SOAP may actually contain any mix of element types that are the derived types of the root type. The use of polymorphic types is indicated by the WSDL and schema descriptions of a Web service and can therefore be predicted/expected for each particular case.

On the one hand, the **typedef** construct provides a convenient way to associate C/C++ types with XML schema types and makes it easy to incorporate these types in a (legacy) C/C++ application. However, on the other hand the **typedef** declarations cannot be used to support polymorphic XML schema types. Most SOAP clients and services do not use polymorphic types. In case they do, the primitive polymorphic types can be declared as a hierarchy of C++ **class**es that can be used simultaneously with the **typedef** declarations.

The general form of a primitive type declaration that is derived from a super type is:

```
class xsd__type_name: [public xsd__super_type_name]
{ public: Type __item;
  [public:] [private] [protected:]
  method1;
  method2;
  ...
};
```

where Type is a primitive C type that may be declared with a **typedef** to enforce XML schema encoding as with the usual **typedef** conventions used by the gSOAP compiler.

For example, the XML schema type hierarchy can be copied to C++ with the following declarations:

```
class xsd__anyType { };
class xsd__anySimpleType: public xsd__anyType { };
typedef char *xsd__anyURI;
class xsd__anyURI_: public xsd__anySimpleType { public: xsd__anyURI __item; };
typedef bool xsd__boolean;
class xsd__boolean_: public xsd__anySimpleType { public: xsd__boolean __item; };
typedef char *xsd__date;
class xsd__date_: public xsd__anySimpleType { public: xsd__date __item; };
typedef time_t xsd__dateTime;
class xsd__dateTime_: public xsd__anySimpleType { public: xsd__dateTime __item; };
typedef double xsd__double;
class xsd__double_: public xsd__anySimpleType { public: xsd__double __item; };
typedef char *xsd__duration;
class xsd__duration_: public xsd__anySimpleType { public: xsd__duration __item; };
typedef float xsd__float;
class xsd__float_: public xsd__anySimpleType { public: xsd__float __item; };
typedef char *xsd__time;
class xsd__time_: public xsd__anySimpleType { public: xsd__time __item; };
typedef char *xsd__decimal;
class xsd__decimal_: public xsd__anySimpleType { public: xsd__decimal __item; };
typedef char *xsd__integer;
class xsd__integer_: public xsd__decimal_ { public: xsd__integer __item; };
typedef LONG64 xsd__long;
```

```
class xsd__long_: public xsd__integer_ { public: xsd__long __item; };
typedef long xsd__int;
class xsd__int_: public xsd__long_ { public: xsd__int __item; };
typedef short xsd__short;
class xsd__short_: public xsd__int_ { public: xsd__short __item; };
typedef char xsd__byte;
class xsd__byte_: public xsd__short_ { public: xsd__byte __item; };
typedef char *xsd__nonPositiveInteger;
class xsd__nonPositiveInteger_: public xsd__integer_ { public: xsd__nonPositiveInteger __item; };
typedef char *xsd__negativeInteger;
class xsd__negativeInteger_: public xsd__nonPositiveInteger_ { public: xsd__negativeInteger __item;
};
typedef char *xsd__nonNegativeInteger;
class xsd__nonNegativeInteger_: public xsd__integer_ { public: xsd__nonNegativeInteger __item; };
typedef char *xsd__positiveInteger;
class xsd__positiveInteger_: public xsd__nonNegativeInteger_ { public: xsd__positiveInteger __item;
};
typedef ULONG64 xsd__unsignedLong;
class xsd__unsignedLong_: public xsd__nonNegativeInteger_ { public: xsd__unsignedLong __item;
};
typedef unsigned long xsd__unsignedInt;
class xsd__unsignedInt_: public xsd__unsginedLong_ { public: xsd__unsignedInt __item; };
typedef unsigned short xsd__unsignedShort;
class xsd__unsignedShort_: public xsd__unsignedInt_ { public: xsd__unsignedShort __item; };
typedef unsigned char xsd__unsignedByte;
class xsd__unsignedByte_: public xsd__unsignedShort_ { public: xsd__unsignedByte __item; };
typedef char *xsd__string;
class xsd__string_: public xsd__anySimpleType { public: xsd__string __item; };
typedef char *xsd__normalizedString;
class xsd__normalizedString_: public xsd__string_ { public: xsd__normalizedString __item; };
typedef char *xsd__token;
class xsd__token_: public xsd__normalizedString_ { public: xsd__token __item; };
```

Note the use of the trailing underscores for the **class** names to distinhuish the **typedef** type names from the **class** names. Only the most frequently used built-in schema types are shown. It is also allowed to include the xsd:base64Binray and xsd:hexBinary types in the hierarchy:

```
class xsd__base64Binary: public xsd__anySimpleType { public: unsigned char *__ptr; int __size;
};
class xsd__hexBinary: public xsd__anySimpleType { public: unsigned char *__ptr; int __size; };
```

See Sections 8.9 and 8.10.

Methods are allowed to be added to the classes above, such as constructors and getter/setter methods.

8.2.3 XML Schema Type Decoding Rules

The decoding rules for the primitive C and C++ data types is given in the table below:

Type	Allows Decoding of	Precision Lost?
bool	[xsd:]boolean	no
char* (C string)	any type, see 8.2.5	no
wchar_t * (wide string)	any type, see 8.2.5	no
double	[xsd:]double	no
	[xsd:]float	no
	[xsd:]long	no
	[xsd:]int	no
	[xsd:]short	no
	[xsd:]byte	no
	[xsd:]unsignedLong	no
	[xsd:]unsignedInt	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
	[xsd:]decimal	possibly
	[xsd:]integer	possibly
	[xsd:]positiveInteger	possibly
	[xsd:]negativeInteger	possibly
	[xsd:]nonPositiveInteger	possibly
	[xsd:]nonNegativeInteger	possibly
float	[xsd:]float	no
	[xsd:]long	no
	[xsd:]int	no
	[xsd:]short	no
	[xsd:]byte	no
	[xsd:]unsignedLong	no
	[xsd:]unsignedInt	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
	[xsd:]decimal	possibly
	[xsd:]integer	possibly
	[xsd:]positiveInteger	possibly
	[xsd:]negativeInteger	possibly
	[xsd:]nonPositiveInteger	possibly
	[xsd:]nonNegativeInteger	possibly
long long	[xsd:]long	no
	[xsd:]int	no
	[xsd:]short	no
	[xsd:]byte	no
	[xsd:]unsignedLong	possibly
	[xsd:]unsignedInt	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
	[xsd:]integer	possibly
	[xsd:]positiveInteger	possibly
	[xsd:]negativeInteger	possibly
	[xsd:]nonPositiveInteger	possibly
	[xsd:]nonNegativeInteger	possibly

Type	Allows Decoding of	Precision Lost?
long	[xsd:]long	possibly, if **long** is 32 bit
	[xsd:]int	no
	[xsd:]short	no
	[xsd:]byte	no
	[xsd:]unsignedLong	possibly
	[xsd:]unsignedInt	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
int	[xsd:]int	no
	[xsd:]short	no
	[xsd:]byte	no
	[xsd:]unsignedInt	possibly
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
short	[xsd:]short	no
	[xsd:]byte	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
char	[xsd:]byte	no
	[xsd:]unsignedByte	possibly
unsigned long long	[xsd:]unsignedLong	no
	[xsd:]unsignedInt	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
	[xsd:]positiveInteger	possibly
	[xsd:]nonNegativeInteger	possibly
unsigned long	[xsd:]unsignedLong	possibly, if **long** is 32 bit
	[xsd:]unsignedInt	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
unsigned int	[xsd:]unsignedInt	no
	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
unsigned short	[xsd:]unsignedShort	no
	[xsd:]unsignedByte	no
unsigned char	[xsd:]unsignedByte	no
time_t	[xsd:]dateTime	no(?)

Due to limitations in representation of certain primitive C++ types, a possible loss of accuracy may occur with the decoding of certain XML schema types as is indicated in the table. The table does not indicate the possible loss of precision of floating point values due to the textual representation of floating point values in SOAP.

All explicitly declared XML schema encoded primitive types adhere to the same decoding rules. For example, the following declaration:

```
typedef unsigned long long xsd__nonNegativeInteger;
```

enables the encoding and decoding of `xsd:nonNegativeInteger` XML schema types (although decoding takes place with a possible loss of precision). The declaration also allows decoding of `xsd:positiveInteger` XML schema types, because of the storage as a **unsigned long long** data type.

8.2.4 Multi-Reference Strings

If more than one **char** pointer points to the same string, the string is encoded as a multi-reference value. Consider for example

```
char *s = "hello", *t = s;
```

The s and t variables are assigned the same string, and when serialized, t refers to the content of s:

```
<string id="123" xsi:type="string">hello</string>
...
<string href="#123"/>
```

The example assumed that s and t are encoded as independent elements.

Note: the use of **typedef** to declare a string type such as xsd__string will not affect the multi-reference string encoding. However, strings declared with different **typedef**s will never be considered multi-reference even when they point to the same string. For example

```
typedef char *xsd__string;
typedef char *xsd__anyURI;
xsd__anyURI *s = "http://www.myservice.com";
xsd__string *t = s;
```

The variables s and t point to the same string, but since they are considered different types their content will not be shared in the SOAP payload through a multi-referenced string.

8.2.5 "Smart String" Mixed-Content Decoding

The implementation of string decoding in gSOAP allows for mixed content decoding. If the SOAP payload contains a complex data type in place of a string, the complex data type is decoded in the string as plain XML text.

For example, suppose the getInfo remote method returns some detailed information. The remote method is declared as:

```
// Contents of header file "getInfo.h":
getInfo(char *detail);
```

The proxy of the remote method is used by a client to request a piece of information and the service responds with:

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: nnn

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
<SOAP-ENV:Body>
<getInfoResponse>
<detail>
<picture>Mona Lisa by <i>Leonardo da Vinci</i></picture>
</detail>
</getInfoResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

As a result of the mixed content decoding, the detail string contains "`<picture>Mona Lisa by <i>Leonardo da Vinci</i></picture>`".

8.2.6 Changing the Encoding Precision of float and double Types

The double encoding format is by default set to "%.18G" (see a manual on printf text formatting in C), i.e. at most 18 digits of precision to limit a loss in accuracy. The float encoding format is by default "%.9G", i.e. at most 9 digits of precision.

The encoding format of a double type can be set by assigning a format string to soap.double_format, where soap is a variable that contains the current runtime environment. For example:

```
struct soap soap;
soap_init(&soap); // sets double_format = "%.18G"
soap.double_format = "%e"; // redefine
```

which causes all doubles to be encoded in scientific notation. Likewise, the encoding format of a float type can be set by assigning a format string to the static soap_float_format string variable. For example:

```
struct soap soap;
soap_init(&soap); // sets float_format = "%.9G"
soap.float_format = "%.4f"; // redefine
```

which causes all floats to be encoded with four digits precision.

Caution: The format strings are not automatically reset before or after SOAP communications. An error in the format string may result in the incorrect encoding of floating point values.

8.2.7 INF, -INF, and NaN Values of float and double Types

The gSOAP runtime stdsoap2.cpp and header file stdsoap2.h support the marshalling of IEEE INF, -INF, and NaN representations. Under certain circumstances this may break if the hardware and/or C/C++ compiler does not support these representations. To remove the representations, remove the inclusion of the ¡math.h¿ header file from the stdsoap2.h file. You can control the representations as well, which are defined by the macros:

```
#define FLT_NAN
#define FLT_PINFTY
#define FLT_NINFTY
#define DBL_NAN
#define DBL_PINFTY
#define DBL_NINFTY
```

8.3 Enumeration Type Encoding and Decoding

Enumerations are generally useful for the declaration of named integer-valued constants, also called enumeration constants.

8.3.1 Symbolic Encoding of Enumeration Constants

The gSOAP stub and skeleton compiler encodes the constants of enumeration-typed variables in symbolic form using the names of the constants when possible to comply to SOAP's XML schema enumeration encoding style. Consider for example the following enumeration of weekdays:

```
enum weekday {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

The enumeration-constant Mon, for example, is encoded as

```
<weekday xsi:type="weekday">Mon</weekday>
```

The value of the xsi:type attribute is the enumeration-type identifier's name. If the element is independent as in the example above, the element name is the enumeration-type identifier's name.

The encoding of complex types such as enumerations requires a reference to an XML schema through the use of a namespace prefix. The namespace prefix can be specified as part of the enumeration-type identifier's name, with the usual namespace prefix conventions for identifiers. This can be used to explicitly specify the encoding style. For example:

```
enum ns1__weekday {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

The enumeration-constant Sat, for example, is encoded as:

```
<ns1:weekday xsi:type="ns1:weekday">Sat</ns1:weekday>
```

The corresponding XML schema for this enumeration data type would be:

```
<xsd:element name="weekday" type="tns:weekday"/>
<xsd:simpleType name="weekday">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Mon"/>
    <xsd:enumeration value="Tue"/>
    <xsd:enumeration value="Wed"/>
    <xsd:enumeration value="Thu"/>
    <xsd:enumeration value="Fri"/>
    <xsd:enumeration value="Sat"/>
    <xsd:enumeration value="Sun"/>
  </xsd:restriction>
</xsd:simpleType>
```

8.3.2 Literal Encoding of Enumeration Constants

If the value of an enumeration-typed variable has no corresponding named constant, the value is encoded as a signed integer literal. For example, the following declaration of a workday enumeration type lacks named constants for Saturday and Sunday:

```
enum ns1__workday {Mon, Tue, Wed, Thu, Fri};
```

If the constant 5 (Saturday) or 6 (Sunday) is assigned to a variable of the workday enumeration type, the variable will be encoded with the integer literals 5 and 6, respectively. For example:

```
<ns1:workday xsi:type="ns1:workday">5</ns1:workday>
```

Since this is legal in C++ and SOAP allows enumeration constants to be integer literals, this method ensures that non-symbolic enumeration constants are correctly communicated to another party if the other party accepts literal enumeration constants (as with the gSOAP stub and skeleton compiler).

Both symbolic and literal enumeration constants can be decoded.

To enforce the literal enumeration constant encoding and to get the literal constants in the WSDL file, use the following trick:

```
enum ns1__nums { _1 = 1, _2 = 2, _3 = 3 };
```

The difference with an enumeration type without a list of values and the enumeration type above is that the enumeration constants will appear in the WSDL service description.

8.3.3 Initialized Enumeration Constants

The gSOAP compiler supports the initialization of enumeration constants, as in:

```
enum ns1__relation {LESS = -1, EQUAL = 0, GREATER = 1};
```

The symbolic names LESS, EQUAL, and GREATER will appear in the SOAP payload for the encoding of the ns1__relation enumeration values -1, 0, and 1, respectively.

8.3.4 How to "Reuse" Symbolic Enumeration Constants

A well-known deficiency of C and C++ enumeration types is the lack of support for the reuse of symbolic names by multiple enumerations. That is, the names of all the symbolic constants defined by an enumeration cannot be reused by another enumeration. To force encoding of the same symbolic name by different enumerations, the identifier of the symbolic name can end in an underscore (_) or any number of underscores to distinghuish it from other symbolic names in C++. This guarantees that the SOAP encoding will use the same name, while the symbolic names can be distinghuished in C++. Effectively, the underscores are removed from a symbolic name prior to encoding.

Consider for example:

```
enum ns1__workday {Mon, Tue, Wed, Thu, Fri};
enum ns1__weekday {Mon_, Tue_, Wed_, Thu_, Fri_, Sat_, Sun_};
```

which will result in the encoding of the constants of `enum ns1__weekday` without the underscore, for example as `Mon`.

Caution: The following declaration:

```
enum ns1__workday {Mon, Tue, Wed, Thu, Fri};
enum ns1__weekday {Sat = 5, Sun = 6};
```

will not properly encode the `weekday` enumeration, because it lacks the named constants for `workday` in its enumeration list.

8.3.5 Boolean Enumeration Type Encoding and Decoding for C Compilers

When a pure C compiler is used to create SOAP clients and services, the `bool` type may not be supported by the compiler and in that case an enumeration type should be used. The C enumeration-type encoding adopted by the gSOAP stub and skeleton compiler can be used to encode boolean values according to the SOAP encoding style. The namespace prefix can be specified with the usual namespace prefix convention for identifiers to explicitly specify the encoding style. For example, the built-in `boolean` XML schema type supports the mathematical concept of binary-valued logic. The `boolean` XML schema encoding style can be specified by using the `xsd` prefix. For example:

```
enum xsd__boolean {false_, true_};
```

The value `false_`, for example, is encoded as:

```
<xsd:boolean xsi:type="xsd:boolean">false</xsd:boolean>
```

Peculiar of the SOAP boolean type encoding is that it only defines the values `0` and `1`, while the built-in XML schema boolean type also defines the `false` and `true` symbolic constants as valid values. The following example declaration of an enumeration type lacks named constants altogether to force encoding of the enumeration values as literal constants:

```
enum SOAP_ENC__boolean {};
```

The value 0, for example, is encoded with an integer literal:

```
<SOAP-ENC:boolean xsi:type="SOAP-ENC:boolean">0<SOAP-ENC:boolean>
```

8.3.6 Bitmask Enumeration Encoding and Decoding

A bitmask is an enumeration of flags such as declared with C#'s [Flags] **enum** annotation. gSOAP supports bitmask encoding and decoding for interoperability. However, bitmask types are not standardized with SOAP RPC.

A special syntactic convention is used in the header file input to the gSOAP compiler to indicate the use of bitmasks with an asterisk:

```
enum * name { enum-constant, enum-constant, ... };
```

The gSOAP compiler will encode the enumeration constants as flags, i.e. as a series of powers of 2 starting with 1. The enumeration constants can be or-ed to form a bitvector (bitmask) which is encoded and decoded as a list of symbolic values in SOAP. For example:

```
enum * ns__machineStatus { ON, BELT, VALVE, HATCH};
int ns__getMachineStatus(char *name, char *enum ns__machineStatus result);
```

Note that the use of the **enum** does not require the asterisk, only the definition. The gSOAP compiler generates the enumeration:

```
enum ns__machineStatus { ON=1, BELT=2, VALVE=4, HATCH=8};
```

A remote method implementation in a Web service can return:

```
int ns__getMachineStatus(struct soap *soap, char *name, enum ns__machineStatus result)
{ ...
  *result = BELT — HATCH;
  return SOAP_OK;
}
```

8.4 struct Encoding and Decoding

A **struct** data type is encoded as a SOAP compound data type such that the **struct** name forms the data type's element name and schema type and the fields of the **struct** are the data type's accessors. This encoding is identical to the **class** instance encoding without inheritance and method declarations, see Section 8.5 for further details. However, the encoding and decoding of **struct**s is more efficient compared to **class** instances due to the lack of inheritance and the requirement by the marshalling routines to check inheritance properties at run time.

8.5 class Instance Encoding and Decoding

A **class** instance is encoded as a SOAP compound data type such that the **class** name forms the data type's element name and schema type and the data member fields are the data type's accessors. Only the data member fields are encoded in the SOAP payload. Class methods are not encoded.

The general form of a **class** declaration is:

```
class [namespace_prefix__]class_name1 [:[public:] [private:] [protected:] [namespace_prefix__]class_name2]
{
  [public:] [private:] [protected:]
  field1;
  field2;
  ...
  [public:] [private:] [protected:]
  method1;
  method2;
  ...
};
```

where

namespace_prefix__ is the optional namespace prefix of the compound data type (see identifier translation rules 7.3)

class_name1 is the element name of the compound data type (see identifier translation rules 7.3).

class_name2 is an optional base class.

field is a field declaration (data member). A field MAY be declared **static** and **const** and MAY be initialized.

method is a method declaration. A method MAY be declared **virtual**, but abstract methods are not allowed. The method parameter declarations are REQUIRED to have parameter identifier names.

[public:] [private:] [protected:] are OPTIONAL and have no effect on the declaration and MAY therefore be ommitted. All access permissions are converted to **public** by the gSOAP stub and skeleton compiler.

A class name is REQUIRED to be unique and cannot have the same name as a **struct**, **enum**, or remote method name specified in the header file input to the gSOAP compiler. The reason is that remote method requests are encoded similarly to class instances in SOAP and they are in principle undistinghuishable (the method parameters are encoded just as the fields of a **class**).

Only single inheritance is supported by the gSOAP compiler. Multiple inheritance is not supported, because of the limitations of the SOAP protocol.

If a constructor method is present, there MUST also be a constructor declaration with empty parameter list.

Templates are not supported by the gSOAP compiler.

A **class** instance is encoded as:

```
<[namespace-prefix:]class-name xsi:type="[namespace-prefix:]class-name">
<basefield-name1 xsi:type="...">...</basefield-name1>
<basefield-name2 xsi:type="...">...</basefield-name2>
...
<field-name1 xsi:type="...">...</field-name1>
<field-name2 xsi:type="...">...</field-name2>
...
</[namespace-prefix:]class-name>
```

where the `field-name` accessors have element-name representations of the class fields and the `basefield-name` accessors have element-name representations of the base class fields. (The optional parts resulting from the specification are shown enclosed in [].)

The decoding of a class instance allows any ordering of the accessors in the SOAP payload. However, if a base class field name is identical to a derived class field name because the field is overloaded, the base class field name MUST precede the derived class field name in the SOAP payload for decoding. gSOAP guarantees this, but interoperability with other SOAP implementations is cannot be guaranteed.

8.5.1 Example

The following example declares a base class ns__Object and a derived class ns__Shape:

```
// Contents of file "shape.h":
class ns__Object
{
  public:
  char *name;
};
class ns__Shape : public ns__Object
{
  public:
  int sides;
  enum ns__Color {Red, Green, Blue} color;
  ns__Shape();
  ns__Shape(int sides, enum ns__Green color);
  ~ns__Shape();
};
```

The implementation of the methods of **class** ns__Shape must not be part of the header file and need to be defined elsewhere.

An instance of **class** ns__Shape with name Triangle, 3 sides, and color Green is encoded as:

```
<ns:Shape xsi:type="ns:Shape">
<name xsi:type="string">Triangle</name>
<sides xsi:type="int">3</sides>
<color xsi:type="ns:Color">Green</color>
</ns:shape>
```

The namespace URI of the namespace prefix `ns` must be defined by a namespace mapping table, see Section 7.4.

8.5.2 Initialized static const Fields

A data member field of a class declared as **static const** is initialized with a constant value at compile time. This field is encoded in the serialization process, but is not decoded in the deserialization process. For example:

```
// Contents of file "triangle.h":
class ns__Triangle : public ns__Object
{
  public:
  int size;
  static const int sides = 3;
};
```

An instance of **class** ns__Triangle is encoded in SOAP as:

```
<ns:Triangle xsi:type="ns:Triangle">
<name xsi:type="string">Triangle</name>
<size xsi:type="int">15</size>
<sides xsi:type="int">3>/sides>
</ns:Triangle>
```

Decoding will ignore the sides field's value.

Caution: The current gSOAP implementation does not support encoding **static const** fields, due to C++ compiler compatibility differences. This feature may be provided the future.

8.5.3 Class Methods

A **class** declaration in the header file input to the gSOAP compiler MAY include method declarations. The method implementations MUST NOT be part of the header file but are required to be defined in another C++ source that is externally linked with the application. This convention is also used for the constructors and destructors of the **class**.

Dynamic binding is supported, so a method MAY be declared **virtual**.

8.5.4 Polymorphism, Derived Classes, and Dynamic Binding

Interoperability between client and service applications developed with gSOAP is established even when clients and/or services use derived classes instead of the base classes used in the declaration of the remote method parameters. A client application MAY use pointers to instances of derived classes for the input parameters of a remote method. If the service was compiled with a declaration and implementation of the derived class, the remote method base class input parameters are demarshalled and a derived class instance is created instead of a base class instance. If the service did not include a declaration of the derived class, the derived class fields are ignored and a base class instance is created. Therefore, interoperability is guaranteed even when the client sends an instance of a derived classes and when a service returns an instance of a derived class.

The following example declares Base and Derived classes and a remote method that takes a pointer to a Base class instance and returns a Base class instance:

```
// Contents of file "derived.h"
class Base
{
  public:
  char *name;
  Base();
  virtual void print();
};
class Derived : public Base
{
  public:
  int num;
  Derived();
  virtual void print();
};
int method(Base *in, struct methodResponse { Base *out; } &result);
```

This header file specification is processed by the gSOAP compiler to produce the stub and skeleton routines which are used to implement a client and service. The pointer of the remote method is also allowed to point to Derived class instances and these instances will be marshalled as Derived class instances and send to a service, which is in accord to the usual semantics of parameter passing in C++ with dynamic binding.

The Base and Derived class method implementations are:

```
// Method implementations of the Base and Derived classes:
#include "soapH.h"
...
Base::Base()
{
  cout << "created a Base class instance" << endl;
}
Derived::Derived()
{
  cout << "created a Derived class instance" << endl;
}
Base::print() {
  cout << "print(): Base class instance " << name << endl;
}
Derived::print() {
  cout << "print(): Derived class instance " << name << " " << num << endl;
}
```

Below is an example CLIENT application that creates a Derived class instance that is passed as the input parameter of the remote method:

```
// CLIENT
#include "soapH.h"
int main()
{
  struct soap soap;
  soap_init(&soap);
  Derived obj1;
```

```
   Base *obj2;
   struct methodResponse r;
   obj1.name = "X";
   obj1.num = 3;
   soap_call_method(&soap, url, action, &obj1, r);
   r.obj2->print();
}
...
```

The following example SERVER1 application copies a class instance (Base or Derived class) from the input to the output parameter:

```
// SERVER1
#include "soapH.h"
int main()
{
   soap_serve(soap_new());
}
int method(struct soap *soap, Base *obj1, struct methodResponse &result)
{
   obj1->print();
   result.obj2 = obj1;
   return SOAP_OK;
}
...
```

The following messages are produced by the CLIENT and SERVER1 applications:

```
CLIENT: created a Derived class instance
SERVER1: created a Derived class instance
SERVER1: print(): Derived class instance X 3
CLIENT: created a Derived class instance
CLIENT: print(): Derived class instance X 3
```

Which indicates that the derived class kept its identity when it passed through SERVER1. Note that instances are created both by the CLIENT and SERVER1 by the demarshalling process.

Now suppose a service application is developed that only accepts Base class instances. The header file is:

```
// Contents of file "base.h":
class Base
{
   public:
   char *name;
   Base();
   virtual void print();
};
int method(Base *in, Base *out);
```

This header file specification is processed by the gSOAP stub and skeleton compiler to produce skeleton routine which is used to implement a service (so the client will still use the derived classes).

The method implementation of the Base class are:

```
// Method implementations of the Base class:
#include "soapH.h"
...
Base::Base() {
   cout << "created a Base class instance" << endl;
}
Base::print()
{
   cout << "print(): Base class instance " << name << endl;
}
```

And the SERVER2 application is that uses the Base class is:

```
// SERVER2
#include "soapH.h"
int main()
{
   soap_serve(soap_new());
}
int method(struct soap *soap, Base *obj1, struct methodResponse &result) {
   obj1->print();
   result.obj2 = obj1;
   return SOAP_OK;
}
...
```

Here are the messages produced by the CLIENT and SERVER2 applications:

```
CLIENT: created a Derived class instance
SERVER2: created a Base class instance
SERVER2: print(): Base class instance X
CLIENT: created a Base class instance
CLIENT: print(): Base class instance X
```

In this example, the object was passed as a Derived class instance to SERVER2. Since SERVER2 only implements the Base class, this object is converted to a Base class instance and send back to CLIENT.

8.6 Pointer Encoding and Decoding

The serialization of a pointer to a data type amounts to the serialization of the data type in SOAP and the SOAP encoded representation of a pointer to the data type is indistinghuishable from the encoded representation of the data type pointed to.

8.6.1 Multi-Reference Data

A data structure pointed to by more than one pointer is serialized as SOAP multi-reference data. This means that the data will be serialized only once and identified with a unique `id` attribute. The encoding of the pointers to the shared data is done through the use of `href` attributes to refer to the multi-reference data (also see Section 6.6 on options to control the serialization of multi-reference data). Cyclic C/C++ data structures are encoded with multi-reference SOAP encoding. Consider for example the following a linked list data structure:

```
typedef char *xsd__string;
struct ns__list
{
  xsd__string value;
  struct ns__list *next;
};
```

Suppose a cyclic linked list is created. The first node contains the value "`abc`" and points to a node with value "`def`" which in turn points to the first node. This is encoded as:

```
<ns:list id="1" xsi:type="ns:list">
  <value xsi:type="xsd:string">abc</value>
  <next xsi:type="ns:list">
    <value xsi:type="xsd:string">def</value>
    <next href="#1"/>
  </next>
</ns:list>
```

In case multi-referenced data is received that "does not fit in a pointer-based structure", the data is copied. For example, the following two **struct**s are similar, except that the first uses pointer-based fields while the other uses non-pointer-based fields:

```
typedef long xsd__int;
struct ns__record
{
  xsd__int *a;
  xsd__int *b;
} P;
struct ns__record
{
  xsd__int a;
  xsd__int b;
} R;
...
  P.a = &n;
  P.b = &n;
...
```

Since both `a` and `b` fields of `P` point to the same integer, the encoding of `P` is multi-reference:

```
<ns:record xsi:type="ns:record">
  <a href="#1"/>
  <b href="#1"/>
</ns:record>
<id id="1" xsi:type="xsd:int">123</id>
```

Now, the decoding of the content in the `R` data structure that does not use pointers to integers results in a copy of each multi-reference integer. Note that the two **struct**s resemble the same XML data type because the trailing underscore will be ignored in XML encoding and decoding.

8.6.2 NULL Pointers and Nil Elements

A NULL pointer is **not** serialized, unless the pointer itself is pointed to by another pointer (but see Section 6.6 to control the serialization of NULLs). For example:

```
struct X
{
  int *p;
  int **q;
}
```

Suppose pointer q points to pointer p and suppose p=NULL. In that case the p pointer is serialized as

```
<... id="123" xsi:nil="true"/>
```

and the serialization of q refers to `href="#123"`. Note that SOAP 1.1 does not support pointer to pointer types (!), so this encoding is specific to gSOAP. The pointer to pointer encoding is rarely used in codes anyway. More common is a pointer to a data type such as a **struct** with pointer fields.

Caution: When the deserializer encounters an XML element that has a `xsi:nil="true"` attribute but the corresponding C++ data is not a pointer or reference, the deserializer will terminate with a SOAP_NULL fault when the soap.enable_null flag is set. The types section of a WSDL description contains information on the "nilability" of data.

8.7 Fixed-Size Arrays

Fixed size arrays are encoded as per SOAP 1.1 one-dimensional array types. Multi-dimensional fixed size arrays are encoded by gSOAP as nested one-dimensional arrays in SOAP. Encoding of fixed size arrays supports partially transmitted and sparse array SOAP formats.

The decoding of (multi-dimensional) fixed-size arrays supports the SOAP multi-dimensional array format as well as partially transmitted and sparse array formats.

An example:

```
// Contents of header file "fixed.h":
struct Example
{
  float a[2][3];
};
```

This specifies a fixed-size array part of the **struct** Example. The encoding of array a is:

```
<a xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="float[][2]">
<SOAP-ENC:Array xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="float[3]"
<float xsi:type="float">...</float>
<float xsi:type="float">...</float>
<float xsi:type="float">...</float>
</SOAP-ENC:Array>
<SOAP-ENC:Array xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="float[3]"
```

```
<float xsi:type="float">...</float>
<float xsi:type="float">...</float>
<float xsi:type="float">...</float>
</SOAP-ENC:Array>
</a>
```

Caution: Any decoded parts of a (multi-dimensional) array that do not "fit" in the fixed size array are ignored by the deserializer.

8.8 Dynamic Arrays

As the name suggests, dynamic arrays are much more flexible than fixed-size arrays and dynamic arrays are better adaptabe to the SOAP encoding and decoding rules for arrays. In addition, a typical C application allocates a dynamic array using malloc, assigns the location to a pointer variable, and deallocates the array later with free. A typical C++ application allocates a dynamic array using new, assigns the location to a pointer variable, and deallocates the array later with delete. Such dynamic allocations are flexible, but pose a problem for the serialization of data: how does the array serializer know the length of the array to be serialized given only a pointer to the sequence of elements? The application stores the size information somewhere. This information is crucial for the array serializer and has to be made explicitly known to the array serializer by packaging the pointer and array size information within a **struct** or **class**.

8.8.1 One-Dimensional Dynamic Arrays

A special form of **struct** or **class** is used for one-dimensional dynamic arrays that contains a pointer variable and a field that records the number of elements the pointer points to in memory.

The general form of the **struct** declaration for one-dimensional dynamic arrays is:

```
struct some_name
{
  Type *__ptr;
  int __size;
  [[static const] int __offset [= ...];]
  ... // anything that follows here will be ignored
};
```

where Type MUST be a type associated with an XML schema or MUST be a primitive type. If these conditions are not met, a list/vector (de)serialization is used (see Section 8.8.6). A primitive type can be used with or without a **typedef**. If the array elements are structs or classes, then the struct/class type names should have a namespace prefix for schema association, or they should be other (nested) dynamic arrays.

An alternative to a **struct** is to use a **class** with optional methods that MUST appear after the __ptr and __size fields:

```
class some_name
{
  public:
  Type *__ptr;
```

```
   int __size;
   [[static const] int __offset [= ...];]
   method1;
   method2;
   ... // any fields that follow will be ignored
};
```

To encode the data type as an array, the name of the **struct** or **class** SHOULD NOT have a namespace prefix, otherwise the data type will be encoded and decoded as a SOAP list/vector, see Section 8.8.6.

The deserializer of a dynamic array can decode partially transmitted and/or SOAP sparse arrays, and even multi-dimensional arrays which will be collapsed into a one-dimensional array with row-major ordering.

8.8.2 Example

The following example header file specifies the XMethods Service Listing service getAllSOAPServices remote method and an array of SOAPService data structures:

```
// Contents of file "listing.h":
class ns3__SOAPService
{
   public:
   int ID;
   char *name;
   char *owner;
   char *description;
   char *homepageURL;
   char *endpoint;
   char *SOAPAction;
   char *methodNamespaceURI;
   char *serviceStatus;
   char *methodName;
   char *dateCreated;
   char *downloadURL;
   char *wsdlURL;
   char *instructions;
   char *contactEmail;
   char *serverImplementation;
};
class ServiceArray
{
   public:
   ns3__SOAPService *__ptr; // points to array elements
   int __size; // number of elements pointed to
   ServiceArray();
   ~ServiceArray();
   void print();
};
int ns__getAllSOAPServices(ServiceArray &return_);
```

An example client application:

```
#include "soapH.h" ...
// ServiceArray class method implementations:
ServiceArray::ServiceArray()
{
  __ptr = NULL;
  __size = 0;
}
ServiceArray::~ServiceArray()
{
  if (__ptr)
    free(__ptr);
  __size = 0;
}
void ServiceArray::print()
{
  for (int i = 0; i ¡ __size; i++)
    cout << __ptr[i].name << ": " << __ptr[i].homepage << endl;
}
...
// Request a service listing and display results:
{
  struct soap soap;
  ServiceArray result;
  const char *endpoint = "www.xmethods.net:80/soap/servlet/rpcrouter";
  const char *action = "urn:xmethodsServicesManager#getAllSOAPServices";
  ...
  soap_init(&soap);
  soap_call_ns__getAllSOAPServices(&soap, endpoint, action, result);
  result.print();
  ...
}
```

8.8.3 One-Dimensional Dynamic Arrays With Non-Zero Offset

The declaration of a dynamic array as described in 8.8 MAY include an **int** __offset field. When set to an integer value, the serializer of the dynamic array will use this field as the start index of the array and the SOAP array offset attribute will be used in the SOAP payload.

For example, the following header file declares a mathematical Vector class, which is a dynamic array of floating point values with an index that starts at 1:

```
// Contents of file "vector.h":
typedef float xsd__float;
class Vector
{
  xsd__float *__ptr;
  int __size;
  int __offset;
  Vector();
  Vector(int n);
  float& operator[](int i);
}
```

The implementations of the Vector methods are:

```
Vector::Vector()
{
   __ptr = NULL;
   __size = 0;
   __offset = 1;
}
Vector::Vector(int n)
{
   __ptr = (float*)malloc(n*sizeof(float));
   __size = n;
   __offset = 1;
}
Vector::~Vector()
{
   if (__ptr)
      free(__ptr);
}
float& Vector::operator[](int i)
{
   return __ptr[i-__offset];
}
```

An example program fragment that serializes a vector of 3 elements:

```
struct soap soap;
soap_init(&soap);
Vector v(3);
v[1] = 1.0;
v[2] = 2.0;
v[3] = 3.0;
soap_begin(&soap);
v.serialize(&soap);
v.put("vec");
soap_end(&soap);
```

The output is a partially transmitted array:

```
<vec xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:float[4]" SOAP-ENC:offset="[1]">
<item xsi:type="xsd:float">1.0</item>
<item xsi:type="xsd:float">2.0</item>
<item xsi:type="xsd:float">3.0</item>
</vec>
```

Note that the size of the encoded array is necessarily set to 4 and that the encoding omits the non-existent element at index 0.

The decoding of a dynamic array with an __offset field is more efficient than decoding a dynamic array without an __offset field, because the __offset field will be assigned the value of the `SOAP-ENC:offset` attribute instead of padding the initial part of the array with default values.

8.8.4 Nested One-Dimensional Dynamic Arrays

One-dimensional dynamic arrays MAY be nested. For example, using **class** Vector declared in the previous section, **class** Matrix is declared:

```
// Contents of file "matrix.h":
class Matrix
{
  public:
  Vector *__ptr;
  int __size;
  int __offset;
  Matrix();
  Matrix(int n, int m);
  ~Matrix();
  Vector& operator[](int i);
};
```

The Matrix type is essentially an array of pointers to arrays which make up the rows of a matrix. The encoding of the two-dimensional dynamic array in SOAP will be in nested form.

8.8.5 Multi-Dimensional Dynamic Arrays

The general form of the **struct** declaration for K-dimensional (K¿1) dynamic arrays is:

```
struct some_name
{
  Type *__ptr;
  int __size[K];
  int __offset[K];
  ... // anything that follows here will be ignored
};
```

where Type MUST be a type associated with an XML schema, which means that it must be a **typedef**ed type in case of a primitive type, or a **struct**/**class** name with a namespace prefix for schema association, or another dynamic array. If these conditions are not met, a list/vector (de)serialization is used (see Section 8.8.6).

An alternative is to use a **class** with optional methods:

```
class some_name
{
  public:
  Type *__ptr;
  int __size[K];
  int __offset[K];
  method1;
  method2;
  ... // any fields that follow will be ignored
};
```

In the above, K is a constant denoting the number of dimensions of the multi-dimensional array.

To encode the data type as an array, the name of the **struct** or **class** SHOULD NOT have a namespace prefix, otherwise the data type will be encoded and decoded as a SOAP list/vector, see Section 8.8.6.

The deserializer of a dynamic array can decode partially transmitted multi-dimensional arrays.

For example, the following declaration specifies a matrix class:

```
typedef double xsd__double;
class Matrix
{
  public:
  xsd__double *__ptr;
  int __size[2];
  int __offset[2];
};
```

In contrast to the matrix class of Section 8.8.4 that defined a matrix as an array of pointers to matrix rows, this class has one pointer to a matrix stored in row-major order. The size of the matrix is determined by the __size field: __size[0] holds the number of rows and __size[1] holds the number of columns of the matrix. Likewise, __offset[0] is the row offset and __offset[1] is the columns offset.

8.8.6 Dynamic Array as List Encoding

In case the name of the **struct** or **class** of a dynamic array has a namespace prefix, the data type is considered a list (a.k.a. vector) and will be serialized as a SOAP list and not encoded as a SOAP array.

For example:

```
struct ns__Map
{
  struct ns__Binding {char *key; char *val;} *__ptr;
  int __size;
};
```

This declares a dynamic array, but the array will be serialized and deserialized as a list. For example:

```
<ns:Map xsi:type="ns:Map">
<ns:Binding xsi:type="ns:Binding">
<key>Joe</key>
<val>555 77 1234</val>
</ns:Binding>
<ns:Binding xsi:type="ns:Binding">
<key>Susan</key>
<val>555 12 6725</val>
</ns:Binding>
<ns:Binding xsi:type="ns:Binding">
<key>Pete</key>
<val>555 99 4321</val>
```

```
</ns:Binding>
</ns:Map>
```

Deserialization is less efficient compared to an array, because the size of the list is not part of the SOAP encoding. Internal buffering is used by the deserializer to collect the elements. When the end of the list is reached, the buffered elements are copied to a newly allocated space on the heap for the dynamic array.

A list (de)serialization is also in affect for dynamic arrays when the pointer field does not refer to a type that is associated with a schema. For example:

```
struct vector
{
  int *__ptr;
  int __size;
};
```

Since **int** has no association with a schema, a vector structure X is serialized as:

```
<X>
<item>1</item>
<item>-2</item>
...
</X>
```

8.8.7 Polymorphic Dynamic Arrays and Lists

An array of pointers to class instances allows the encoding of polymorphic arrays (arrays of polymorphic element types) and lists. For example:

```
class ns__Object
{
  public:
  ...
};
class ns__Data: public ns__Object
{
  public:
  ...
};
class ArrayOfObject
{
  public:
  ns__Object **__ptr;
  int __size;
  int __offset;
};
```

The pointers in the array can point to the ns__Object base class or ns__Data derived class instances which will be serialized and deserialized accordingly in SOAP. That is, the array elements are polymorphic.

8.8.8 How to Change the Tag Names of the Elements of a SOAP Array or List

The __ptr field in a **struct** or **class** declaration of a dynamic array may have an optional suffix part that describes the name of the tags of the SOAP array XML elements. The suffix is part of the field name:

```
Type *__ptrarray_elt_name
```

The suffix describes the tag name to be used for all array elements. The usual identifier to XML translations apply, see Section 7.3. The default XML element tag name for array elements is `item` (which corresponds to the use of field name __ptritem).

Consider for example:

```
struct ArrayOfstring
{
  xsd__string *__ptrstring;    int __size; };
```

The array is serialized as:

```
<array xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[2]">
<string xsi:type="xsd:string">Hello</string>
<string xsi:type="xsd:string">World</string>
</array>
```

SOAP 1.1 and 1.2 do not require the use of a specific tag name for array elements. gSOAP will deserialize a SOAP array while ignoring the tag names. Certain XML schemas used in doc/literal encoding may require the declaration of arrray element tag names.

8.8.9 Embedded Arrays and Lists

An array (or list) can be embedded in a struct/class without the need to declare a separate array data type. When a struct or class type declaration contains a **int** __size field and the next field below is a pointer type, gSOAP assumes the pointer type points to an array of values where the __size field holds the number of values at run time. Multiple arrays can be embedded in a struct/class by using __size field names that end with a unique name suffix.

The general convention for embedding arrays is:

```
struct ns__SomeStruct
{
  ...
  int __sizename1; // number of elements pointed to
  Type1 *field1; // by this field
  ...
  int __sizename2; // number of elements pointed to
  Type2 *field2; // by this field
  ...
};
```

where name1 and name2 are identifiers used as a suffix to distinghuish the __size field. These names can be arbitrary and are not visible in XML.

For example, the following struct has two embedded arrays:

```
struct ns__Contact
{
  char *firstName;
  char *lastName;
  int __sizePhones;
  ULONG64 *phoneNumber; // array of phone numbers
  int __sizeEmails;
  char **emailAddress; // array of email addresses
  char *socSecNumber;
};
```

The XML serialization of an example ns__Contact is:

```
<mycontact xsi:type="ns:Contact">
  <firstName>Joe</firstName>
  <lastName>Smith</lastName>
  <phoneNumber>5551112222</phoneNumber>
  <phoneNumber>5551234567</phoneNumber>
  <phoneNumber>5552348901</phoneNumber>
  <emailAddress>Joe.Smith@mail.com</emailAddress>
  <emailAddress>Joe@Smith.com</emailAddress>
  <socSecNumber>999999999</socSecNumber>
</mycontact>
```

8.9 Base64Binary XML Schema Type Encoding

The base64Binary XML schema type is a special form of dynamic array declared with a pointer (__ptr) to an **unsigned char** array.

For example using a **struct**:

```
struct xsd__base64Binary
{
  unsigned char *__ptr;
  int __size;
};
```

Or with a **class**:

```
class xsd__base64Binary
{
  public:
  unsigned char *__ptr;
  int __size;
};
```

When compiled by the gSOAP stub and skeleton compiler, this header file specification will generate base64Binary serializers and deserializers.

The SOAP_ENC:base64 encoding is another type for base 64 binary encoding specified by the SOAP data type schema and some SOAP applications may use this form (as indicated by their WSDL descriptions). It is declared by:

```
struct SOAP_ENC__base64
{
   unsigned char *__ptr;
   int __size;
};
```

Or with a **class**:

```
class SOAP_ENC__base64
{
   unsigned char *__ptr;
   int __size;
};
```

When compiled by the gSOAP stub and skeleton compiler, this header file specification will generate SOAP-ENC:base64 serializers and deserializers.

The advantage of using a **class** is that methods can be used to initialize and manipulate the __ptr and __size fields. The user can add methods to this class to do this. For example:

```
class xsd__base64Binary
{
   public:
   unsigned char *__ptr;
   int __size;
   xsd__base64Binary(); // Constructor
   xsd__base64Binary(int n); // Constructor
   ~xsd__base64Binary(); // Destructor
   unsigned char *location(); // returns the memory location
   int size(); // returns the number of bytes
};
```

Here are example method implementations:

```
xsd__base64Binary::xsd__base64Binary()
{
   __ptr = NULL;
   __size = 0;
}
xsd__base64Binary::xsd__base64Binary(int n)
{
   __ptr = (unsigned int*)malloc(n);
   __size = n;
}
xsd__base64Binary::~xsd__base64Binary()
{
   if (__ptr)
      free(__ptr);
}
unsigned char *xsd__base64Binary::location()
{
  return __ptr;
}
int xsd__base64Binary::size()
{
  return __size;
}
```

The following example in C/C++ reads from a raw image file and encodes the image in SOAP using the `base64Binary` type:

```
...
FILE *fd = fopen("image.jpg", "r");
xsd__base64Binary image(filesize(fd));
fread(image.location(), image.size(), 1, fd);
fclose(fd);
soap_begin(&soap);
image.soap_serialize(&soap);
image.soap_put(&soap, "jpegimage", NULL);
soap_end(&soap);
...
```

where filesize is a function that returns the size of a file given a file descriptor.

Reading the `xsd:base64Binary` encoded image.

```
...
xsd__base64Binary image;
soap_begin(&soap);
image.get(&soap, "jpegimage");
soap_end(&soap);
...
```

The **struct** or **class** name soap_enc__base64 should be used for `SOAP-ENC:base64` schema type instead of xsd__base64Binary.

8.10 hexBinary XML Schema Type Encoding

The `hexBinary` XML schema type is a special form of dynamic array declared with the name xsd__hexBinary and a pointer (__ptr) to an **unsigned char** array.

For example, using a **struct**:

```
struct xsd__hexBinary
{
  unsigned char *__ptr;
  int __size;
};
```

Or using a **class**:

```
class xsd__hexBinary
{
  public:
  unsigned char *__ptr;
  int __size;
};
```

When compiled by the gSOAP stub and skeleton compiler, this header file specification will generate `base64Binary` serializers and deserializers.

8.11 Doc/Literal XML Encoding Style

gSOAP supports doc/literal SOAP encoding of request and/or response messages. However, there are some limitations on the XML format to support (de)serialization of XML documents into C/C++ data structures. XML documents may contain constructs that gSOAP cannot parse and will simply ignore because the gSOAP XML parser has been optimized for SOAP data. This occurs when the XML document uses XML attribute values that are part of the data. Arbitrary XML documents can be (de)serialized into regular C strings or wide character strings (wchar_t*) by gSOAP. Because XML documents are stored in strings, an application may need a "plug-in" XML parser to decode XML content stored in strings. For details on (de)serialization XML into strings, see Section 8.11.1.

gSOAP supports doc/literal SOAP encoding either manually by setting soap.encodingStyle, soap.defaultNamespace, and soap.disable_href of the current gSOAP environment soap, or automatically by using a gSOAP directive in the header file. In most doc/literal cases, the `SOAP-ENV:encodingStyle` attribute needs to be absent. To do this, set soap.encodingStyle=NULL. Furthermore, a default namespace needs to be defined by setting soap.defaultNamespace. Finally, doc/literal is a limited form of serialization and does not support graphs. So setting soap.disable_href=1 will not produce multi-reference data. Note that cyclic data will crash the doc/literal serializer because of this setting. Also polymorphic data may cause deserialization problems due to the absense of type information in the SOAP payload (which makes us wonder why doc/literal is the default in .NET).

The LocalTimeByZipCode remote method of the LocalTime service provides the local time given a zip code and uses doc/literal SOAP encoding (using MS .NET). The following header file declares the method:

```
int LocalTimeByZipCode(char *ZipCode, char **LocalTimeByZipCodeResult);
```

Note that none of the data types need to be namespace qualified using namespace prefixes. The use of namespace prefixes is optional for doc/literal in gSOAP. When used, the XML document will include `xsi:type` attributes.

To illustrate the manual doc/literal setting, the following client program sets the required properties before the call:

```
#include "soapH.h"
int main()
{
  struct soap soap;
  char *t;
```

```
   soap_init(&soap);
   soap.encodingStyle = NULL; // don't use SOAP encoding
   soap.defaultNamespace = "http://alethea.net/webservices/"; // use the service's namespace
   soap.disable_href = 1;" // don't produce multi-ref data (but can accept)
   if (soap_call_LocalTimeByZipCode(&soap, "http://alethea.net/webservices/LocalTime.asmx", "http://alethea.net/w
"32306", &t))
      soap_print_fault(&soap, stderr);
   else
      printf("Time = %s\n", t);
   return 0;
}
struct Namespace namespaces[] =
{
   {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
   {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"},
   {"xsi", "http://www.w3.org/2001/XMLSchema-instance", "http://www.w3.org/*/XMLSchema-
instance"},
   {"xsd", "http://www.w3.org/2001/XMLSchema", "http://www.w3.org/*/XMLSchema"},
   {NULL, NULL}
};
```

The SOAP request is:

```
POST /webservices/LocalTime.asmx HTTP/1.0
Host: alethea.net
Content-Type: text/xml; charset=utf-8
Content-Length: 479
SOAPAction: "http://alethea.net/webservices/LocalTimeByZipCode"

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://alethea.net/webservices/">
  <SOAP-ENV:Body>
    <LocalTimeByZipCode><ZipCode>32306</ZipCode></LocalTimeByZipCode>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Alternatively, the settings can be automatically set by including gSOAP directives in the header file:

```
//gsoap ns service name: localtime
//gsoap ns service encoding: literal
//gsoap ns service namespace: http://alethea.net/webservices/
int ns__LocalTimeByZipCode(char *ZipCode, char **LocalTimeByZipCodeResult);
```

In this case, the method name requires to be associated with a schema through a namespace prefix, e.g. ns is used in this example. See Section 12.2 for more details on gSOAP directives. With these directives, the gSOAP compiler generates client and server sources with the specified settings. The directives are required to produce a WSDL file for a new service that uses doc/literal encoding.

The example client program can be simplified into:

```
#include "soapH.h"
#include "localtime.nsmap" // include generated map file
int main()
{
   struct soap soap;
   char *t;
   soap_init(&soap);
   if (soap_call_ ns__LocalTimeByZipCode(&soap, "http://alethea.net/webservices/LocalTime.asmx",
"http://alethea.net/webservices/LocalTimeByZipCode", "32306", &t))
      soap_print_fault(&soap, stderr);
   else
      printf("Time = %s\n", t);
   return 0;
}
```

8.11.1 Serializing and Deserializing XML Into Strings

To declare a literal XML "type" to hold XML documents in regular strings, use:

```
typedef char *XML;
```

To declare a literal XML "type" to hold XML documents in wide character strings, use:

```
typedef wchar_t *XML;
```

Note: only one of the two storage formats can be used. The differences between the use of regular strings versus wide character strings for XML documents are:

- Regular strings for XML documents MUST hold UTF-8 encoded XML documents. That is, the string MUST contain the proper UTF-8 encoding to exchange the XML document in SOAP messages.
- Wide character strings for XML documents SHOULD NOT hold UTF-8 encoded XML documents. Instead, the UTF-8 translation is done automatically by the gSOAP runtime marshalling routines.

Literal XML encoding should only use one input parameter and one output parameter. Here is an example of a remote method specification in which the parameters of the remote method uses literal XML encoding to pass an XML document to a service and back:

```
typedef char *XML;
ns__GetDocument(XML m__XMLDoc, XML &m__XMLDoc_);
```

The ns__Document is essentially a **struct** that forms the root of the XML document. The use of the underscore in the ns__Document response part of the message avoids the name clash between the **struct**s. Assuming that the namespace mapping table contains the binding of ns to `http://my.org/` and the binding of m to `http://my.org/mydoc.xsd`, the XML message is:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:ns="http://my.org/"
  xmlns:m="http://my.org/mydoc.xsd"
  SOAP-ENV:encodingStyle="">
  <SOAP-ENV:Body>
    <ns:GetDocument>
      <XMLDoc xmlns="http://my.org/mydoc.xsd">
        ...
      </XMLDoc>
    </ns:Document>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Important: the literal XML encoding style MUST be specified by setting soap.encodingStyle, where soap is a variable that contains the current runtime environment. For example, to specify no constraints on the encoding style (which is typical) use NULL:

```
struct soap soap;
soap_init(&soap);
soap.encodingStyle = NULL;
```

As a result, the `SOAP-ENV:encodingStyle` attribute will not appear in the SOAP payload.

For interoperability with Apache SOAP, use

```
struct soap soap;
soap_init(&soap);
soap.encodingStyle = "http://xml.apache.org/xml-soap/literalxml";
```

The name of the response element can be changed (default is the remote method name ending with Response). For example:

```
typedef char *XML;
ns__GetDocument(struct soap *soap, XML m__XMLDoc, struct ns__Document { XML m__XMLDoc;
} &result);
```

9 SOAP Fault Processing

A predeclared standard SOAP Fault data structure is generated by the gSOAP stub and skeleton compiler for exchanging exception messages. This predeclared data structure is:

```
struct SOAP_ENV__Fault
{
  char *faultcode;
  char *faultstring;
  char *faultactor;
```

```
   char *detail;
};
```

The data structure can be changed to the need of an application. To do this, include a new declaration of a **struct** SOAP_ENV__Fault in the header file input to the gSOAP compiler to replace the built-in data structure. A **class** for the SOAP Fault data structure is not allowed in gSOAP 2.0. For example:

```
struct SOAP_ENV__Fault
{
   char *faultcode; // MUST be string
   char *faultstring; // MUST be string
   char *faultactor;
   char *detail; // MUST be string
   Detail *t__detail; // new detail field (note namespace prefix "t")
};
```

where DetailType is some data type that holds application specific data such as a stack dump.

When the proxy of a remote method returns an error (see Section 7.2), thne soap.fault contains the SOAP Fault data.

When a remote method wants to raise an exception, it does so by assigning the fault field of the current reference to the runtime environment with appropriate data associated with the exception and by returning the error SOAP_FAULT. For example:

```
   soap_fault(soap); // allocate fault struct if necessary
   soap->fault->faultstring = "Stack dump";
   soap->fault->detail = NULL;
   soap->fault->t__detail = sp; // point to stack (needs stack serializer)
   return SOAP_Fault; // return from remote method call
```

When soap_fault allocates a fault struct, this data can be removed with the soap_end call (or soap_dealloc).

Each remote method implementation in a service application can return a SOAP Fault upon an exception by returning an error code, see Section 5.2.1 for details and an example. In addition, a SOAP Fault can be returned by a service application through calling the soap_send_fault function. This is useful in case the initialization of the application fails, as illustrated in the example below:

```
int main()
{
   struct soap soap;
   soap_init(&soap);
   some initialization code
   if (initialization failed)
   {
      soap.error = SOAP_FAULT;
      soap_fault(&soap);
      soap.fault->faultcode = "Server";
      soap.fault->faultstring = "Init failed";
      soap.fault->details = "...";
      soap_send_fault(&soap); // Send SOAP Fault to client
```

```
    return 0; // Terminate
  }
}
```

10 SOAP Header Processing

A predeclared standard SOAP Header data structure is generated by the gSOAP stub and skeleton compiler for exchanging SOAP messages with SOAP Headers. This predeclared data structure is:

```
struct SOAP_ENV__Header
{ void *dummy;
};
```

which declares and empty header (some C and C++ compilers don't accept empty structs so a transient dummy field is provided).

To adapt the data structure to a specific need for SOAP Header processing, a new **struct** SOAP_ENV__Header can be added to the header file input to the gSOAP compiler. A **class** for the SOAP Header data structure can be used instead of a **struct**.

For example, the following header can be used for transaction control:

```
struct SOAP_ENV__Header
{ char *t__transaction;
};
```

with client-side code:

```
  struct soap soap;
  soap_init(&soap);
...
soap.header = NULL; // do not use a SOAP Header for the request (as set with soap_init)
soap.actor = NULL; // do not use an actor (receiver is actor)
soap_call_method(&soap, ...);
if (soap.header) // a SOAP Header was received
  cout << soap.header->t__transaction;
// Can reset, modify, or set soap.header here before next call
soap_call_method(&soap, ...); // reuse the SOAP Header of the service response for the request
...
```

The SOAP Web service response can include a SOAP Header with a transaction number that the client is supposed to use for the next remote method invocation to the service. Therefore, the next request includes a transaction number:

```
...
<SOAP-ENV:Envelope ...>
<SOAP-ENV:Header>
<t:transaction xsi:type="int">12345</t:transaction>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
...
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

This is just an example and the transaction control is not a feature of SOAP but can be added on by the application layer to implement stateful transactions between clients and services. At the client side, the soap.actor attribute can be set to indicate the recipient of the header (the SOAP `SOAP-ENV:actor` attribute).

A Web service can read and set the SOAP Header as follows:

```
int main()
{
   struct soap soap;
   soap.actor = NULL; // use this to accept all headers (default)
   soap.actor = "http://some/actor"; // accept headers destined for "http://some/actor" only
   soap_serve(&soap);
}
...
int method(struct soap *soap, ...)
{
   if (soap->header) // a Header was received
      ... = soap->header->t__transaction;
   else
      soap->header = soap_malloc(sizeof(struct SOAP_ENV__Header)); // alloc new header
...        soap->header->t__transaction = ...;
   return SOAP_OK;
}
```

See Section 12.2 on how to generate WSDL with the proper method-to-header-part bindings.

The `SOAP-ENV:mustUnderstand` attribute indicates the requirement that the recipient of the SOAP Header (who must correspond to the `SOAP-ENV:actor` attribute when present or when `SOAP-ENV:actor="http://s`
MUST handle the Header part that carries the attribute. gSOAP handles this automatically on the background. However, an application still needs to inspect the header part's value and handle it appropriately. If a remote method in a Web service is not able to do this, it should return SOAP_MUSTUNDERSTAND to indicate this failure.

The syntax for the header file input to the gSOAP compiler is extended with a special storage qualifier mustUnderstand. This qualifier can be used in the SOAP Header declaration to indicate which parts should carry a SOAP-ENV:mustUnderstand="1" attrbute. For example:

```
struct SOAP_ENV__Header
{
   char *t__transaction;
   mustUnderstand char *t__authentication;
};
```

When both fields are set and soap.actor="http://some/actor" then the message contains:

```
<SOAP-ENV:Envelope ...>
<SOAP-ENV:Header>
<t:transaction>5</t:transaction>
<t:authentication SOAP-ENV:actor="http://some/actor" SOAP-ENV:mustUnderstand="1">XX</t:authenti
</SOAP-ENV:Header>
<SOAP-ENV:Body>
...
```

```
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

11 DIME Attachment Processing

gSOAP can transmit binary data with DIME attachments. The binary data is stored in augmented `xsd:base64Binary` and `xsd:hexBinary` structs/classes. These structs/classes have three additional fields: an `id` field for attachment referencing (typically a CID or UUID), a `type` field to specify the MIME type of the binary data, and an `options` field to piggy-back additional information with a DIME attachment. DIME attachment support is fully automatic, which means that gSOAP will test for the presence of attachments at run time and use SOAP in DIME accordingly.

A `xsd:base64Binary` type with DIME attachment support is declared by

```
struct xsd__base64Binary
{
  unsigned char *__ptr;
  int __size;
  char *id;
  char *type;
  char *options;
};
```

The specification order of the fields is important. In addition, no other fields or methods may be declared before any of these fields in the struct/class. A `xsd:hexBinary` declaration is similar. If the `id` field and/or `type` field is non-NULL during serialization of the data, DIME attachment transmission is used for the *entire* SOAP message, as per SOAP in DIME specifications. If only the `type` field is set, gSOAP will assign a default DIME id to the attachment (see also below). The `options` field is an optional string with a special layout: the first two bytes are reserved for the option type, the next two bytes store the size of the option data, followed by the option data. The function

```
char *soap_option(struct soap *soap, unsigned short type, const char *option)
```

returns a string with this encoding. For example

```
struct xsd__base64Binary image;
image.__ptr = ...;
image.__size = ...;
image.id = "uuid:09233523-345b-4351-b623-5dsf35sgs5d6";
image.type = "image/jpeg";
image.options = soap_option(soap, 0, "My wedding picture");
```

When receiving DIME attachments, the fields will be set according to the DIME attachment data. If binary data is received without attachments, the `id`, `type`, `options` fields are NULL. Note that SOAP messages may contain binary data that references external resources not provided as attachments. In that case, the `__ptr` field is NULL and the `id` field refers to the external data source. Non-augmented binary data types `xsd:base64Binary` and `xsd:hexBinary` can be used to receive DIME attachments, but the `id`, `type`, and `options` information cannot be processed due to the absence of these fields. Also strings can be received as DIME attachments, but not send.

If necessary, the `xsd:base64Binary` schema type and its attachment-based type can be separated with class inheritance. For example:

```
class xsd__base64Binary
{
  unsigned char *__ptr;
  int __size;
};
class xsd__base64Binary_ : xsd__base64Binary
{
  char *id;
  char *type;
  char *options;
};
```

The dime_id_format attribute of the current gSOAP run-time environment can be set to the default format of DIME id fields. The format string MUST contain a `%d` format specifier (or any other **int**-based format specifier). The value of this specifier is a non-negative integer, with zero being the value of the DIME attachment id for the SOAP message. For example,

```
struct soap soap;
soap_init(&soap);
soap.dime_id_format = "uuid:09233523-345b-4351-b623-5dsf35sgs5d6-%x";
```

As a result, all attachments with a NULL id field will use a gSOAP-generated id value based on the format string.

Caution: Do not set the disable_request_count or disable_response_count attributes of the gSOAP run-time with DIME.

12 Advanced Features

12.1 Internationalization

The use of wide-character strings (wchar_t*) in C and C++ clients and services suffices for internationalization. In contrast, when strings with wide characters are received and stored in regular strings, only the lower 8 bits of the wide characters are retained. The enable_utf_string attribute of the current gSOAP environment can be set to enable send and receive of wide-characters with regular strings. With this attrbute set, text will be stored in UTF8 format in the strings directly, which means that character codes 1 to 127 are treated as plain ASCII. Codes with the MSB set are UTF8-encoded characters. Please consult the UTF8 specification for details.

12.2 Customizing the WSDL and Namespace Mapping Table File Contents

A header file can be augmented with directives for the gSOAP Stub and Skeleton compiler to automatically generate customized WSDL and namespace mapping tables contents. The WSDL and namespace mapping table files do not need to be modified by hand (Sections 5.2.5 and 7.4). These compiler directive are specified as //-comments.

Three directives are currently supported that can be used to specify details associated with namespace prefixes used by the remote method names in the header file. To specify the name of a Web Service in the header file, use:

```
//gsoap namespace-prefix service name: service-name
```

where *namespace-prefix* is a namespace prefix used by identifiers in the header file and *service-name* is the name of a Web Service (only required to create new Web Services).

To specify the location of a Web Service in the header file, use:

```
//gsoap namespace-prefix service location: URL
```

where *URL* is the location of the Web Service (only required to create new Web Services). The *URL* specifies the path to the service executable (so *URL/service-executable* is the actual locations of the executable).

To specify the name of the executable of a Web Service in the header file, use:

```
//gsoap namespace-prefix service executable: executable-name
```

where *executable-name* is the name of the executable of the Web Service.

To specify the namespace URI of a Web Service in the header file, use:

```
//gsoap namespace-prefix service namespace: namespace-URI
```

where *namespace-URI* is the URI associated with the namespace prefix.

In addition, the schema namespace URI can be specified in the header file:

```
//gsoap namespace-prefix schema namespace: namespace-URI
```

where *namespace-URI* is the schema URI associated with the namespace prefix. If present, it affects the schema-part of the generated WSDL file and the URI in the namespace mapping table. This declaration is useful when the service declares it's own data types that need to be associated with a namespace. Furthermore, the header file for client applications do not need the full service details and the specification of the schema namespaces for namespace prefixes suffices.

When header processing is required, each method declared in the WSDL should provide a binding to the parts of the header that may appear as part of a method request message. Such a binding is given by:

```
//gsoap namespace-prefix service method-header-part: method-name header-part
```

For example:

```
struct SOAP_ENV__Header
{
  char *h__transaction;
  struct UserAuth *h__authentication;
};
```

Suppose method ns__login uses both header parts (at most), then this is declared as:

```
//gsoap ns service method-header-part: login transaction
//gsoap ns service method-header-part: login authentication
int ns__login(...);
```

Suppose method ns__search uses only the first header part (at most), then this is declared as:

```
//gsoap ns service method-header-part: search transaction
int ns__search(...);
```

Note that the method name and header part names in the directive are left unqualified.

To specify the header parts for the method input (method request message), use:

```
//gsoap namespace-prefix service method-input-header-part: method-name header-part
```

Similarly, to specify the header parts for the method output (method response message), use:

```
//gsoap namespace-prefix service method-output-header-part: method-name header-part
```

The declarations above only affect the WSDL. It's the application's responsibility to set and reset the header messages.

When doc/literal encoding is required, the service encoding can be specified in the header file:

```
//gsoap namespace-prefix service encoding: literal
```

or when the SOAP-ENV:encodingStyle attribute is different from the SOAP 1.1 encoding style:

```
//gsoap namespace-prefix service encoding: encoding-style
```

(Note: blanks can be used anywhere in the directive, except between // and gsoap.)

The use of these directive is best illustrated with an example. The quotex.h header file of the quotex example in the gSOAP distribution for Unix/Linux is:

```
//gsoap ns1 service namespace: urn:xmethods-delayed-quotes
int ns1__getQuote(char *symbol, float &result);

//gsoap ns2 service namespace: urn:xmethods-CurrencyExchange
int ns2__getRate(char *country1, char *country2, float &result);

//gsoap ns3 service name: quotex
//gsoap ns3 service location: http://www.cs.fsu.edu/~engelen
//gsoap ns3 service namespace: urn:quotex
int ns3__getQuote(char *symbol, char *country, float &result);
```

The quotex example is a new Web Service created by combining two existing Web Services: the XMethods Delayed Stock Quote service and XMethods Currency Exchange service.

Namespace prefix ns3 is used for the new quotex Web Service with namespace URI urn:quotex, service name quotex, and location http://www.cs.fsu.edu/~engelen. Since the new Web Service invokes the ns1__getQuote and ns2__getRate remote methods, the service namespaces of these methods are given. The service names and locations of these methods are not given because they are only required for setting up a new Web Service for these methods (but may also be provided in the header file for documentation purposes). After invoking the gSOAP Stub and Skeleton compiler on the quotex.h header file:

```
soapcpp2 quotex.h
```

the WSDL of the new quotex Web Service is saved as quotex.wsdl. Since the service name (quotex), location (http://www.cs.fsu.edu/~engelen), and namespace URI (urn:quotex) were provided in the header file, the generated WSDL file does not need to be changed by hand and can be published immediately together with the compiled Web Service installed as a CGI application at the designated URL (http://www.cs.fsu.edu/~engelen/quotex.cgi and http://www.cs.fsu.edu/~engelen/quotex.wsdl).

The namespace mapping table for the quotex.cpp Web Service implementation is saved as quotex.nsmap. This file can be directly included in quotex.cpp instead of specified by hand in the source of quotex.cpp:

```
#include "quotex.nsmap"
```

The automatic generation and inclusion of the namespace mapping table requires compiler directives for **all** namespace prefixes to associate each namespace prefix with a namespace URI. Otherwise, namespace URIs have to be manually added to the table (they appear as %{URI}%).

12.3 Transient Data Types

There are situations when certain data types have to be ignored by gSOAP for the compilation of (de)marshalling routines. For example, in certain cases the fields in a class or struct need not be (de)serialized, or the base class of a derived class should not be (de)serialized, and certain built-in classes such as ostream cannot be (de)serialized. These data types (including fields) are called "transient" and can be declared outside of gSOAP's compilation window. Transient data type and transient fields are declared with the **extern** keyword or are declared within [and] blocks in the header file input to the gSOAP compiler. The **extern** keyword has a special meaning to the gSOAP compiler and won't affect the generated codes. The special [and] block construct can be used with data type declarations and within **struct** and **class** declarations. The use of **extern** or [ ] achieve the same effect, but [ ] may be more convenient to encapsulate transient types in a larger part of the header file. The use of **extern** with **typedef** is reserved for the declaration of user-defined external (de)serializers for data types, see Section 12.4.

First example:

```
extern class ostream; // ostream can't be (de)serialized, but need to be declared to make it visible
to gSOAP
class ns__myClass
{ ...
  virtual void print(ostream &s) const; // need ostream here
  ...
};
```

Second example:

```
[
  class myBase // base class need not be (de)serialized
  { ... };
]
class ns__myDerived : myBase
{ ... };
```

Third example:

```
[ typedef int transientInt; ]
class ns__myClass
{
  int a; // will be (de)serialized
  [
  int b; // transient field
  char s[256]; // transient field
  ]
  extern float d; // transient field
  char *t; // will be (de)serialized
  transientInt *n; // transient field
  [
  virtual void method(char buf[1024]); // does not create a char[1024] (de)serializer
  ]
};
```

In this example, **class** ns__myClass has three transient fields: b, s, and n which will not be (de)serialized in SOAP. Field n is transient because the type is declared within a transient block. Pointers, references, and arrays of transient types are transient. The single class method is encapsulated within [and] to prevent gSOAP from creating (de)serializers for the **char**[1024] type. gSOAP will generate (de)serializers for all types that are not declared within a [and] transient block.

Functions prototypes of remote methods cannot be declared transient and will result in errors when attempted.

12.4 How to Declare User-Defined Serializers and Deserializers

Users can declare their own (de)serializers for specific data types instead of relying on the gSOAP-generated (de)serializers. To declare a external (de)serializer, declare a type with **extern typedef**. gSOAP will not generate the (de)serialzers for the type name that is declared. For example:

```
extern typedef char *MyData;
struct Sample
{
  MyData s; // use user-defined (de)serializer for this field
  char *t; // use gSOAP (de)serializer for this field
};
```

The user is required to supply the following routines for each **extern typedef**'ed name T:

```
void soap_mark_T(struct soap *soap, const T *a)
void soap_default_T(struct soap *soap, T *a)
void soap_out_T(struct soap *soap, const char *tag, int id, const T *a, const char *type)
T *soap_in_T(struct soap *soap, const char *tag, T *a, const char *type)
```

The function prototypes can be found in soapH.h.

For example, the (de)serialization of MyData can be done with the following code:

```
void soap_mark_MyData(struct soap *soap, MyData *const*a)
{ } // no need to mark this node (for multi-ref and cycle detection)
void soap_default_MyData(&soap, MyData **a)
{ *a = NULL }
void soap_out_MyData(struct soap *soap, const char *tag, int id, MyData *const*a, const char
*type)
{
  soap_element_begin_out(soap, tag, id, type); // print XML beginning tag
  soap_send(soap, *a); // just print the string (no XML conversion)
  soap_element_end_out(soap, tag); // print XML ending tag
}
MyData **soap_in_MyData(struct soap *soap, const char *tag, MyData **a, const char *type)
{
  if (soap_element_begin_in(soap, tag))
    return NULL;
  if (!a)
    a = (MyData**)soap_malloc(soap, sizeof(MyData*));
  if (soap->null)
    *a = NULL; // xsi:nil element
  if (*soap->type && soap_match_tag(soap, soap->type, type))
  {
    soap->error = SOAP_TYPE_MISMATCH;
    return NULL; // type mismatch
  }
  if (*soap->href)
    a = (MyData**)soap_id_forward(soap, soap-¿href, a, SOAP_MyData, sizeof(MyData*))
  else if (soap->body)
  {
    char *s = soap_value(soap); // fill buffer
    *a = (char*)soap_malloc(soap, strlen(s)+1);
    strcpy(*a, s);
  }
  if (soap-¿body && soap_element_end_in(soap, tag))
    return NULL;
  return a;
```

More information on custom (de)serialization will be provided in this document or in a separate document in the future. The writing of the (de)serializer code requires the use of the low-level gSOAP API.

12.5 How to Serialize Data Without XML `xsi:type` Attributes

gSOAP serializes data in XML with `xsi:type` attributes when the types are declared with namespace prefixes to indicate the type of the data contained in the elements. SOAP 1.1 and 1.2 requires `xsi:type` attributes in the presence of polymorphic data or when the type of the data cannot be deduced from the SOAP payload.

To omit the generation of `xsi:type` attributes in the serialization, simply use type declarations that do not include namespace prefixes. The only remaining issue is the (de)serialization of lists/vectors with typed elements. To declare a list/vector with typed elements, use a leading underscores for type names of the **struct** or **class**. The leading underscores in type names makes type anonymous (invisible in XML).

12.6 Function Callbacks for Customized I/O and HTTP Handling

gSOAP provides five callback functions for customized I/O and HTTP handling:

Callback (function pointer)
int (*soap.fopen)(**struct** soap *soap, **const char** *endpoint, **const char** *host, **int** port) Called from a client proxy to open a connection to a Web Service located at endpoint Input parameters host and port are micro-parsed from endpoint Should return a valid file descriptor, or -1 and soap->error set to an error code Built-in gSOAP function: tcp_connect
int (*soap.fpost)(**struct** soap *soap, **const char** *endpoint, **const char** *host, **const char** *path, **const char** *actio Called from a client proxy to generate the HTTP header to connect to endpoint Input parameters host and path are micro-parsed from endpoint, action is the SOAP action, and count is the length of the SOAP message or 0 when soap.disable_request_count$\neq$0 Use function soap_send(**struct** soap *soap, **char** *s) to write the header contents Should return SOAP_OK, or a gSOAP error code Built-in gSOAP function: http_post
int (*soap.fresponse)(**struct** soap *soap, **int** soap_error_code, size_t count) Called from a service to generate the response HTTP header Input parameter soap_error_code is a gSOAP error code (see Section 7.2 and count is the length of the SOAP message or 0 when soap.disable_response_count$\neq$0 Use function soap_send(**struct** soap *soap, **char** *s) to write the header contents Should return SOAP_OK, or a gSOAP error code Built-in gSOAP function: http_response
int (*soap.fparse)(**struct** soap *soap) Called by client proxy and service to parse an HTTP header (if present) When user-defined, this routine must at least skip the header Use function **int** soap_getline(**struct** soap *soap, **char** *buf, **int** len) to read HTTP header lines into a buffer buf of length len (returns empty line at end of HTTP header) Should return SOAP_OK, or a gSOAP error code Built-in gSOAP function: http_parse
int (*soap.fclose)(**struct** soap *soap) Called by client proxy **multiple times**, to close a socket connection before a new socket connection is established and at the end of communications when soap.keep_alive$\neq$0 Should return SOAP_OK, or a gSOAP error code Built-in gSOAP function: tcp_disconnect
int (*soap.fsend)(**struct** soap *soap, **const char** *s, size_t n) Called for all send operations to emit contents of s of length n Should return SOAP_OK, or a gSOAP error code Built-in gSOAP function: fsend
size_t (*soap.frecv)(**struct** soap *soap, **char** *s, size_t n) Called for all receive operations to fill buffer s of maximum length n Should return the number of bytes read or 0 in case of an error, e.g. EOF Built-in gSOAP function: frecv
int (*soap_fignore)(**struct** soap *soap, **const char** *tag) Called when an unknown XML element was encountered on the input and tag is the offending XML el Should return SOAP_OK, or a gSOAP error code such as SOAP_MUSTUNDERSTAND to throw an e Built-in gSOAP function: fignore

In addition, a **void***user field in the **struct** soap data structure has been reserved that can serve to point to user-defined data that needs to be carried along the callbacks.

The following example uses I/O function callbacks for customized serialization of data into a buffer and deserialization back into a datastructure:

```
char buf[10000]; // XML buffer
int len1 = 0; // #chars written
int len2 = 0; // #chars read
// mysend: put XML in buf[]
int mysend(struct soap *soap, const char *s, size_t n)
{
   if (len1 + n > sizeof(buf))
      return SOAP_EOF;
   strcpy(buf + len1, s);
   len1 += n;
   return SOAP_OK;
}
// myrecv: get XML from buf[]
size_t myrecv(struct soap *soap, char *s, size_t n)
{
   if (len2 + n > len1)
      n = len1 - len2;
   strncpy(s, buf + len2, n);
   len2 += n;
   return n;
}
main()
{
   struct soap soap;
   struct ns__person p;
   soap_init(&soap);
   len1 = len2 = 0; // reset buffer pointers
   p.name = "John Doe";
   p.age = 25;
   soap.fsend = mysend; // assign callback
   soap.frecv = myrecv; // assign callback
   soap_begin(&soap);
   soap.enable_embedding = 1;
   soap_serialize_ns__person(&soap, &p);
   soap_put_ns__person(&soap, &p, "ns:person", NULL);
   if (soap.error)
   {
      soap_print_fault(&soap, stdout);
      exit(-1);
   }
   soap_end(&soap);
   soap_begin(&soap);
   soap_get_ns__person(&soap, &p, "ns:person", NULL);
   if (soap.error)
   {
      soap_print_fault(&soap, stdout);
      exit(-1);
   }
   soap_end(&soap);
   soap_init(&soap); // disable callbacks
}
```

The soap_done function can be called to reset the callback to the default internal gSOAP I/O and HTTP handlers.

The following example illustrates customized I/O and (HTTP) header handling. The SOAP request is saved to a file. The client proxy then reads the file contents as the service response. To perform this trick, the service response has exactly the same structure as the request. This is declared by the **struct** ns__test output parameter part of the remote method declaration. This struct resembles the service request (see the generated soapStub.h file created from the header file).

The header file is:

```
//gsoap ns service name: callback
//gsoap ns service namespace: urn:callback
struct ns__person
{
  char *name;
  int age;
};
int ns__test(struct ns__person in, struct ns__test &out);
```

The client program is:

```
#include "soapH.h"
...
int myopen(struct soap *soap, const char *endpoint, const char *host, int port)
{
  if (strncmp(endpoint, "file:", 5))
  {
    printf("File name expected\n");
    return SOAP_EOF;
  }
  if ((soap->sendfd = soap->recvfd = open(host, O_RDWR, S_IWUSR—S_IRUSR)) ¡ 0)
    return SOAP_EOF;
  return SOAP_OK;
}
void myclose(struct soap *soap)
{
  if (soap->sendfd ¿ 2) // still open?
    close(soap->sendfd); // then close it
  soap->recvfd = 0; // set back to stdin
  soap->sendfd = 1; // set back to stdout
}
int mypost(struct soap *soap, const char *endpoint, const char *host, const char *path, const
char *action, size_t count)
{
  return soap_send(soap, "Custom-generated file\n"); // writes to soap->sendfd
}
int myparse(struct soap *soap)
{
  char buf[256];
  if (lseek(soap->recvfd, 0, SEEK_SET) ¡ 0 —— soap_getline(soap, buf, 256)) // go to begin and
skip custom header
    return SOAP_EOF;
  return SOAP_OK;
}
main()
```

```
{
  structsoap soap;
  struct ns__test r;
  struct ns__person p;
  soap_init(&soap); // reset
  p.name = "John Doe";
  p.age = 99;
  soap.fopen = myopen; // use custom open
  soap.fpost = mypost; // use custom post
  soap.fparse = myparse; // use custom response parser
  soap.fclose = myclose; // use custom close
  soap_call_ns__test(&soap, "file://test.xml", "", p, r);
  if (soap.error)
  {
    soap_print_fault(&soap, stdout);
    exit(-1);
  }
  soap_end(&soap);
  soap_init(&soap); // reset to default callbacks
}
```

SOAP 1.1 and 1.2 specify that XML elements may be ignored when present in a SOAP payload on the receiving side. gSOAP ignores XML elements that are unknown, unless the XML attribute `mustUnderstand="true"` is present in the XML element. It may be undesirable for elements to be ignored when the outcome of the omission is uncertain. The soap.fignore callback can be set to a function that returns SOAP_OK in case the element can be safely ignored, or SOAP_MUSTUNDERSTAND to throw an exception, or to perform some application-specific action. For example, to throw an exception as soon as an unknown element is encountered on the input, use:

```
int myignore(struct soap *soap, const char *tag)
{
  return SOAP_MUSTUNDERSTAND; // never skip elements (secure)
}
...
soap.fignore = myignore;
soap_call_ns__method(&soap, ...); // or soap_serve(&soap);
```

To selectively throw an exception as soon as an unknown element is encountered but element `ns:xyz` can be safely ignored, use:

```
int myignore(struct soap *soap, const char *tag)
{
  if (soap_match_tag(soap, tag, "ns:xyz") != SOAP_OK)
    return SOAP_MUSTUNDERSTAND;
  return SOAP_OK;
}
...
soap.fignore = myignore;
soap_call_ns__method(&soap, ...); // or soap_serve(&soap)
...
struct Namespace namespaces[] =
{
  {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
  {"SOAP-ENC","http://schemas.xmlsoap.org/soap/encoding/"},
  {"xsi", "http://www.w3.org/1999/XMLSchema-instance"},
  {"xsd", "http://www.w3.org/1999/XMLSchema"},
  {"ns", "some-URI"}, // the namespace of element ns:xyz
  {NULL, NULL}
```

Function soap_match_tag compares two tags. The third parameter may be a pattern where * is a wildcard and - is a single character wildcard. So for example soap_match_tag(tag, "ns:*") will match any element in namespace ns or when no namespace prefix is present in the XML message.

The callback can also be used to keep track of unknown elements in an internal data structure such as a list:

```
struct Unknown
{
  char *tag;
  struct Unknown *next;
};
int myignore(struct soap *soap, const char *tag)
{
  char *s = (char*)soap_malloc(soap, strlen(tag)+1);
  struct Unknown *u = (struct Unknown*)soap_malloc(soap, sizeof(struct Unknown));
  if (s && u)
  {
    strcpy(s, tag);
    u-¿tag = s;
    u-¿next = ulist;
    ulist = u;
  }
}
...
struct soap *soap;
struct Unknown *ulist = NULL;
soap_init(&soap);
soap.fignore = myignore;
soap_call_ns__method(&soap, ...); // or soap_serve(&soap)
// print the list of unknown elements
soap_end(&soap); // clean up
```

12.7 HTTP 1.0 and 1.1

gSOAP uses HTTP 1.0 by default. gSOAP supports HTTP 1.1, but does not support all HTTP 1.1 transfer encodings such as gzipped encodings. gSOAP does support HTTP 1.1 chunked-transfer encoding. Nevertheless, the the HTTP version used can be changed by setting the attribute:

```
struct soap soap;
soap_init(&soap);
...
soap.http_version = "1.1";
```

12.8 HTTP Keep-Alive

gSOAP supports keep-alive socket connections. To activate keep-alive support, set the keep_alive attribute:

```
struct soap soap;
soap_init(&soap);
...
soap.keep_alive = 1;
```

When a client proxy communicates with a service that closes the connection, soap.keep_alive will be reset to 0 afterwards.

Keep-alive support can be activated for stand-alone services:

```
main()
{
  ...
  soap.keep_alive = 1;
  s = soap_accept(&soap);
  ...
  while (soap_serve(&soap) == SOAP_OK && soap.keep_alive)
    ...;
  ...
}
```

The connection will be kept open on the server-side only if the request contains an HTTP 1.0 header with "`Connection: Keep-Alive`" or an HTTP 1.1 header that does not contain "`Connection: close`". This means that a gSOAP client method call should use "`http://`" in the endpoint URL of the request. If the client does not close the connection, the server will wait forever when no timeout is specified.

12.9 HTTP Chunked Transfer Encoding

gSOAP supports HTTP chunked transfer encoding. Un-chunking of inbound messages takes place automatically. Outbound messages are never chunked, except when the soap.chunked_transfer=1 is set, where soap is the current gSOAP run-time environment. Most Web services, however, will not accept chunked inbound messages. Note that chunking allows soap.disable_request_count=0 and soap.disable_response_count=0, **but not when DIME attachments are used**.

12.10 Timeout Management for Non-Blocking Operations

Socket connect, accept, send, and receive timeout values can be set to manage socket communication timeouts. The soap.connect_timeout, soap.accept_timeout, soap.send_timeout, and soap.recv_timeout attributes of the current gSOAP runtime environment soap can be set to the appropriate user-defined socket send, receive, and accept timeout values. A positive value measures the timeout in seconds. A negative timeout value measures the timeout in microseconds (10^{-6} sec).

The soap.connect_timeout specifies the timeout value for soap_call_ns__method calls.

The soap.accept_timeout specifies the timeout value for soap_accept(&soap) calls.

The soap.send_timeout and soap.recv_timeout specify the timeout values for non-blocking socket I/O operations.

Example:

```
struct soap soap;
soap_init(&soap);
soap.send_timeout = 10;
soap.recv_timeout = 10;
```

This will result in a timeout if no data can be send in 10 seconds and no data is received within 10 seconds after initiating a send or receive operation over the socket. A value of zero disables timeout, for example:

```
soap.send_timeout = 0;
soap.recv_timeout = 0;
```

When a timeout occurs in send/receive operations, a SOAP_EOF exception will be raised ("end of file or no input"). Negative timeout values measure timeouts in microseconds, for example:

```
#define uSec *-1
#define mSec *-1000
soap.accept_timeout = 10 uSec;
soap.send_timeout = 20 mSec;
soap.recv_timeout = 20 mSec;
```

The macros improve readability.

Caution: Many Linux versions do not support non-blocking connect(). Therefore, setting soap.connect_timeout for non-blocking soap_call_ns__method calls may not work under Linux.

12.11 Secure SOAP Clients with HTTPS/SSL

You need to install the OpenSSL library on your platform to enable secure SOAP clients to utilize HTTPS/SSL. After installation, compile your application with option -DWITH_OPENSSL. For example on Linux:

```
g++ -DWITH_OPENSSL myclient.cpp stdsoap.cpp soapC.cpp soapClient.cpp -lssl -lcrypto
```

or Unix:

```
g++ -DWITH_OPENSSL myclient.cpp stdsoap.cpp soapC.cpp soapClient.cpp -lxnet -lsocket -lnsl
-lssl -lcrypto
```

or you can add the following line to stdsoap.h:

```
#define WITH_OPENSSL
```

A client program simply uses the prefix https: instead of http: in the endpoint URL of a remote method call to a Web Service to use encrypted transfers (if the service supports HTTPS). For example:

```
soap_call_ns__mymethod(&soap, "https://domain/path/secure.cgi", "", ...);
```

By default, server authentication is disabled. To enable server authentication, set the require_server_auth attribute of the current gSOAP runtime environment (**struct** soap) before a call is made:

```
soap.require_server_auth = 1;
```

This will force server authentication for all calls over HTTPS.

Make sure you have signal handlers set in your application to catch broken connections (SIGPIPE):

```
signal(SIGPIPE, sigpipe_handle);
```

where, for example:

```
void sigpipe_handle(int x) { }
```

12.12 Secure SOAP Web Services with HTTPS/SSL

When a Web Service is installed as CGI, it uses standard I/O that is encryped/decrypted by the Web server that runs the CGI application. Therefore, HTTPS/SSL support must be configured for the Web server (not Web Service).

SSL support for stand-alone gSOAP Web services is accomplished by calling soap_ssl_accept after soap_accept. In addition, a key file, CA file, DH file, and password need to be supplied. Instructions on how to do this can be found in the OpenSSL documentation. To enable OpenSSL, first install OpenSSL and use option -DWITH_OPENSSL with your C or C++ compiler, for example:

```
g++ -DWITH_OPENSSL -o myprog myprog.cpp stdsoap2.cpp soapC.cpp soapServer.cpp -lssl -
lcrypto
```

Let's take a look at an example SSL secure multi-threaded stand-alone SOAP Web Service:

```
int main()
{
  int m, s;
  pthread_t tid;
  struct soap soap, *tsoap;
  soap_init(&soap);
  soap.keyfile = "server.pem"; // must be resident key file
  soap.cafile = "cacert.pem"; // must be resident CA file
  soap.dhfile = "dh512.pem"; // must be resident DH file
  soap.password = "password"; // password
  m = soap_bind(&soap, "linprog2.cs.fsu.edu", 18000, 100);
  if (m < 0)
  {
    soap_print_fault(&soap, stderr);
    exit(-1);
  }
  fprintf(stderr, "Socket connection successful: master socket = %d\n", m);
  for (;;)
```

```
    {
      s = soap_accept(&soap);
      fprintf(stderr, "Socket connection successful: slave socket = %d\n", s);
      if (s < 0)
      {
        soap_print_fault(&soap, stderr);
        exit(-1);
      }
      if (soap_ssl_accept(&soap))
      {
        soap_print_fault(&soap, stderr);
        exit(-1);
      }
      tsoap = soap_new();
      if (!tsoap)
        exit(-1);
      tsoap->socket = soap.socket; // set by soap_accept
      tsoap->ssl = soap.ssl; // set by soap_ssl_accept
      tsoap->bio = soap.bio; // set by soap_ssl_accept
      pthread_create(&tid, NULL, &process_request, (void*)tsoap);
    }
    return 0;
  }
  void *process_request(void *soap)
  {
    pthread_detach(pthread_self());
    soap_serve((struct soap*)soap);
    soap_end((struct soap*)soap);
    free(soap);
    return NULL;
  }
```

In case Web services have to verify clients, use a key file, CA file, and password in an SSL-enabled client:

```
...
soap_init(&soap);
soap.keyfile = "client.pem";
soap.password = "password";
soap.cafile = "cacert.pem";
if (soap_call_ns__method(&soap, "https://linprog2.cs.fsu.edu:18000", "", ...)
...
```

Make sure you have signal handlers set in your service and/or client applications to catch broken connections (SIGPIPE):

```
signal(SIGPIPE, sigpipe_handle);
```

where, for example:

```
void sigpipe_handle(int x) { }
```

12.13 Client-Side Cookie Support

Client-side cookie support is optional. To enable cookie support, compile with option -DWITH_COOKIES, for example:

```
g++ -DWITH_COOKIES -o myclient stdsoap2.cpp soapC.cpp soapClient.cpp
```

or add the following line to stdsoap.h:

```
#define WITH_COOKIES
```

Client-side cookie support is fully automatic. So just (re)compile stdsoap2.cpp with -DWITH_COOKIES to enable cookie-based session control in your client.

A database of cookies is kept and returned to the appropriate servers. Cookies are not automatically saved to a file by a client. So the internal cookie database is discarded when the client program terminates.

To avoid "cookie storms" caused by malicious servers that return an unreasonable amount of cookies, gSOAP clients/servers are restricted to a database size that the user can limit (32 cookies by default), for example:

```
struct soap soap;
soap_init(&soap);
soap.cookie_max = 10;
```

The cookie database is a linked list pointed to by soap.cookies where each node is declared as:

```
struct soap_cookie
{
  char *name;
  char *value;
  char *domain;
  char *path;
  long expire; /* client-side: local time to expire; server-side: seconds to expire */
  unsigned int version;
  short secure;
  short session; /* server-side */
  short env; /* server-side: got cookie from client */
  short modified; /* server-side: client cookie was modified */
  struct soap_cookie *next;
};
```

Since the cookie database is linked to a soap struct, each thread has a local cookie database in a multi-threaded implementation.

12.14 Server-Side Cookie Support

Server-side cookie support is optional. To enable cookie support, compile with option -DWITH_COOKIES, for example:

```
g++ -DWITH_COOKIES -o myserver ...
```

gSOAP provides the following cookie API for server-side cookie session control:

Function
struct soap_cookie *soap_set_cookie(**struct** soap *soap, **const char** *name, **const char** *value, **const char** *domain,
Add a cookie to the database with name name and value value. domain and path may be NULL to use the current domain and path given by soap_cookie_domain and s If successful, returns pointer to a cookie node in the linked list, or NULL otherwise.
struct soap_cookie *soap_cookie(**struct** soap *soap, **const char** *name, **const char** *domain, **const char** *path);
Find a cookie in the database with name name and value value. domain and path may be NULL to use the current domain and path given by soap_cookie_domain and s If successful, returns pointer to a cookie node in the linked list, or NULL otherwise.
char *soap_cookie_value(**struct** soap *soap, **const char** *name, **const char** *domain, **const char** *path);
Get value of a cookie in the database with name name. domain and path may be NULL to use the current domain and path given by soap_cookie_domain and s If successful, returns the string pointer to the value, or NULL otherwise.
long soap_cookie_expire(**struct** soap *soap, **const char** *name, **const char** *domain, **const char** *path);
Get expiration value of the cookie in the database with name name (in seconds). domain and path may be NULL to use the current domain and path given by soap_cookie_domain and s Returns the expiration value, or -1 if cookie does not exist.
int soap_set_cookie_expire(**struct** soap *soap, **const char** *name, **long** expire, **const char** *domain, **const char** *pat
Set expiration value expire of the cookie in the database with name name (in seconds). domain and path may be NULL to use the current domain and path given by soap_cookie_domain and s If successful, returns SOAP_OK, or SOAP_EOF otherwise.
int soap_set_cookie_session(**struct** soap *soap, **const char** *name, **const char** *domain, **const char** *path);
Set cookie in the database with name name to be a session cookie. This means that the cookie will be returned to the client. (Only cookies that are modified are returned to the client). domain and path may be NULL to use the current domain and path given by soap_cookie_domain and s If successful, returns SOAP_OK, or SOAP_EOF otherwise.
int soap_clr_cookie_session(**struct** soap *soap, **const char** *name, **const char** *domain, **const char** *path);
Clear cookie in the database with name name to be a session cookie. domain and path may be NULL to use the current domain and path given by soap_cookie_domain and s If successful, returns SOAP_OK, or SOAP_EOF otherwise.
void soap_clr_cookie(**struct** soap *soap, **const char** *name, **const char** *domain, **const char** *path);
Remove cookie from the database with name name. domain and path may be NULL to use the current domain and path given by soap_cookie_domain and s
int soap_getenv_cookies(**struct** soap *soap);
Initializes cookie database by reading the 'HTTP_COOKIE' environment variable. This provides a means for a CGI application to read cookies send by a client. If successful, returns SOAP_OK, or SOAP_EOF otherwise.
void soap_free_cookies(**struct** soap *soap);
Release cookie database.

The following global variables are used to define the current domain and path:

Attribute	**value**
const char *cookie_domain	MUST be set to the domain (host) of the service
const char *cookie_path	MAY be set to the default path to the service
int cookie_max	maximum cookie database size (default=32)

The cookie_path value is used to filter cookies intended for this service according to the path prefix rules outlined in RFC2109.

The following example server adopts cookies for session control:

```
int main()
{
   struct soap soap;
   int m, s;
   soap_init(&soap);
   soap.cookie_domain = "...";
   soap.cookie_path = "/"; // the path which is used to filter/set cookies with this destination
   if (argc < 2)
   {
      soap_getenv_cookies(&soap); // CGI app: grab cookies from 'HTTP_COOKIE' env var
      soap_serve(&soap);
   }
   else
   {
      m = soap_bind(&soap, NULL, atoi(argv[1]), 100);
      if (m < 0)
         exit(-1);
      for (int i = 1; ; i++)
      {
         s = soap_accept(&soap);
         if (s < 0)
            exit(-1);
         soap_serve(&soap);
         soap_end(&soap); // clean up
         soap_free_cookies(&soap); // remove all old cookies from database so no interference occurs
with the arrival of new cookies
      }
   }
   return 0;
}
int ck__demo(struct soap *soap, ...)
{
   int n;
   const char *s;
   s = soap_cookie_value(soap, "demo", NULL, NULL); // cookie returned by client?
   if (!s)
      s = "init-value"; // no: set initial cookie value
   else
      ... // modify 's' to reflect session control
   soap_set_cookie(soap, "demo", s, NULL, NULL);
   soap_set_cookie_expire(soap, "demo", 5, NULL, NULL); // cookie may expire at client-side in 5
seconds
   return SOAP_OK;
}
```

12.15 Connecting Clients Through Proxy Servers

When a client needs to connect to a Web Service through a proxy server, set the soap.proxy_host string and soap.proxy_port integer attributes of the current soap runtime environment to the proxy's host name and port, respectively. For example:

```
struct soap soap;
soap_init(&soap);
soap.proxy_host = "proxyhostname";
soap.proxy_port = 8080;
if (soap_call_ns__method(&soap, "http://host:port/path", "action", ...))
   soap_print_fault(&soap, stderr);
else
   ...
```

The attributes soap.proxy_host and soap.proxy_port keep their values throug the remove method calls, so they only need to be set once.

12.16 FastCGI Support

To enable FastCGI support, install FastCGI and compile with option -DWITH_FASTCGI or add

```
#define WITH_FASTCGI
```

to stdsoap2.h.