

# Using rpcgen

---

This chapter describes the rpcgen compiler. It includes these sections:

- **What rpcgen Does**  
Describes the rpcgen compiler and its input, output, and interfaces.
- **Converting Local Procedures into Remote Procedures**  
Uses a printmessage example to illustrate converting a local procedure to a remote procedure. Describes the RPC steps involved and the steps for completing the conversion process.
- **Generating XDR Routines**  
Provides an example protocol description file and explains XDR routines for converting data types as well as how to test the client and server procedures together.
- **The C Preprocessor**  
Describes the symbols that may be defined and includes a description of rpcgen preprocessing.
- **rpcgen Programming Notes**  
Includes timeout changes, handling broadcast on the server side, and other information passed to server procedures.
- **The RPC Language**  
Describes definitions, structures, unions, enumerations, typedefs, constants, programs, declarations, and special cases.

## What rpcgen Does

The rpcgen compiler exists to help you write RPC applications simply and directly. rpcgen does most of the work, letting you debug the main features of your application, instead of requiring you to spend most of your time debugging your network interface code.

---

**Note** The rpcgen compiler is not supplied with the Cisco IOS for S/390 RPC/XDR product but may be available on remote workstations. It can be useful to generate C language output, which can then be transferred to the mainframe.

---

### How rpcgen Works

rpcgen is a compiler. It accepts a remote program interface definition written in a language, called RPC Language, which is similar to C. It produces a C language output that includes stub versions of the client routines, a server skeleton, XDR filter routines for both parameters and results, and a header file that contains common definitions.

The client stubs interface with the RPC library and effectively hide the network from their callers. The server stub similarly hides the network from the server procedures that are invoked by remote clients. rpcgen's output files can be compiled and linked in the usual way.

You write server procedures and link them with the server skeleton, produced by rpcgen, to get an executable server program. To use a remote program, you write an ordinary main program that makes local procedure calls to the client stubs produced by rpcgen. Linking this program with rpcgen's stubs creates an executable program. (At present the main program must be written in C). rpcgen options can be used to suppress stub generation and to specify the transport to be used by the server stub.

Like all compilers, rpcgen reduces development time that would otherwise be spent coding and debugging low-level routines at a small cost in efficiency and flexibility. In addition, like many compilers, rpcgen allows escape hatches for programmers to mix low-level code with high-level code. In speed-critical applications, hand-written routines can be linked with the rpcgen output without any difficulty. Also, you may proceed by using rpcgen output as a starting point, and then rewriting it as necessary. (For a discussion of RPC programming without rpcgen, see Using rpcgen.)

## Converting Local Procedures into Remote Procedures

One task that may need to be done is to convert an application that runs on a single machine to one that runs over the network.

### A printmessage Example

This example—a program, printmessage, that prints a message to the console—is converted so that a message can be sent to the console from anywhere in the system:

```
/* printmsg.c: print a message on the console */

#include <stdio.h>

main(argc, argv)
    int argc;
    char *argv[];
{
    char *message;

    if (argc != 2)
    {
        fprintf(stderr, "usage: %s <message>\n", argv[0]);
        exit(1);
    }
    message = argv[1];

    if (!printmessage(message))
    {
        fprintf(stderr, "%s: couldn't print your message\n", argv[0]);
        exit(1);
    }
    printf("Message Delivered!\n");
    exit(0);
}
```

---

```

    }

    /* Print a message to the console. Return a boolean indicating
     * whether the message was actually printed. */

printmessage(msg)
    char *msg;
{
    FILE *f;

    f = fopen("/dev/console", "w");
    if (f == NULL)
    {
        return (0);
    }
    fprintf(f, "%s\n", msg);
    fclose(f);
    return(1);
}

```

And then, of course:

```

example% cc printmsg.c -o printmsg
example% printmsg "Hello, there."
Message delivered!
example%

```

## Remote Procedures Steps

If `printmessage` is turned into a remote procedure, it can be called from anywhere in the network. Ideally, you would only insert a keyword, like `remote`, in front of a procedure to turn it into a remote procedure.

### Determine Procedure Input and Output Types

You must first determine what types there are for all procedure inputs and outputs. In this example, `printmessage` takes a string as input, and returns an integer as output. Knowing this, you write a protocol specification like this in RPC language that describes the remote version of `printmessage`:

```

/* msg.x: Remote message printing protocol */

program MESSAGEPROG
{
    version MESSAGEVERS
    {
        int PRINTMESSAGE(string) = 1;
    } = 1;
} = 99;

```

Remote procedures are part of remote programs, so an entire remote program is actually declared here that contains the single procedure `PRINTMESSAGE`. This procedure was declared to be in version 1 of the remote program. No null procedure (procedure 0) is necessary because `rpcgen` generates it automatically.

In this example, the argument type is `string` and not `char *`. This is because a `char *` in C is ambiguous. Programmers usually intend it to mean a null-terminated string of characters, but it could also represent a pointer to a single character or a pointer to an array of characters. In RPC language, a null-terminated string is unambiguously called a `string`.

### The Remote Procedure

You next write the actual remote procedure. Here is the definition of a remote procedure to implement the PRINTMESSAGE procedure previously declared:

```
/* msg_proc.c: implementation of the remote procedure "printmessage" */

#include <stdio.h>
#include <rpc.h>          /* always needed */
#include "msg.h"          /* msg.h will be generated by rpcgen */

/* Remote version of "printmessage" */

int *
printmessage_1(msg)
    char **msg;
{
    static int result;      /* must be static! */
    FILE *f;
    f = fopen("/dev/console", "w");
    if (f == NULL)
    {
        result = 0;
        return (&result);
    }
    fprintf(f, "%s\n", *msg);
    fclose(f);
    result = 1;
    return (&result);
}
```

The declaration of the remote procedure `printmessage_1` differs from that of the local procedure `printmessage` in these ways:

- It takes a pointer to a string instead of a string itself. This is true of all remote procedures: they always take pointers to their arguments rather than the arguments themselves.
- It returns a pointer to an integer instead of an integer itself. This is also generally true of remote procedures: they always return a pointer to their results.
- It has a `_1` appended to its name. In general, all remote procedures called by `rpcgen` are named by this rule: the name in the program definition (here `PRINTMESSAGE`) is converted to all lower-case letters, an underscore (`_`) is appended to it, and the version number (here `1`) is appended.

### Declare the Main Client Program

The last step is to declare the main client program that calls the remote procedure. Here is an example:

```
/* rprintmsg.c: remote version of "printmsg.c" */
#include <stdio.h>
#include <rpc.h>          /* always needed */
#include "msg.h"          /* msg.h will be generated by rpcgen */

main(argc, argv)
    int argc;
    char *argv[];
{
    CLIENT *cl;
    int *result;
    char *server;
    char *message;
```

---

```

if (argc != 3)
{
    fprintf(stderr,
        "usage: %s host message\n", argv[0]);
    exit(1);
}

/* Save values of command line arguments */

server = argv[1];
message = argv[2];

/* Create client "handle" used for calling MESSAGEPROG on the
 * server designated on the command line. We tell the RPC
 * package to use the "tcp" protocol when contacting
 * the server. */

cl = clnt_create(server, MESSAGEPROG, MESSAGEVERS, "tcp");
if (cl == NULL)
{
    /* *Couldn't establish connection with server.
     * Print error message and die. */

    clnt_pcreateerror (server);
    exit(1);
}

/* Call the remote procedure "printmessage" on the server */

result = printmessage_1(&message, cl);
if (result == NULL)
{
    /* An error occurred while calling the server.
     * Print error message and die. */

    clnt_perror (cl, server);
    exit(1);
}

/* Okay, we successfully called the remote procedure. */

if (*result == 0)
{
    /* Server was unable to print our message.
     * Print error message and die. */
    fprintf(stderr, "%s: %s couldn't print your message\n",
        argv[0], server);
    exit(1);
}

/* The message got printed on the server's console */

printf("Message delivered to %s!\n", server);
exit(0);
}

```

The client handle (called `handle` in the example) used by `rpcgen` is created using the RPC library routine `clnt_create`. This client handle is passed to the stub routines that call the remote procedure.

The remote procedure `printmessage_1` is called exactly the same way as it is declared in `msg_proc.c` except for the inserted client handle as the first argument.

## Completing the Process

Here is how to complete the process:

```
example% rpcgen msg.x
example% cc rprintmsg.c msg_clnt.c -o rprintmsg
example% cc msg_proc.c msg_svc.c -o msg_server
```

Two programs were compiled:

- The client program `rprintmsg`
- The server program `msg_server`.

Before doing this, `rpcgen` was used to fill in the missing pieces.

Here is what `rpcgen` did with the input file `msg.x`:

- It created a header file called `msg.h` that contained `#defines` for `MESSAGEPROG`, `MESSAGEVERS` and `PRINTMESSAGE` for use in the other modules.
- It created client stub routines in the `msg_clnt.c` file. In this case there is only one, the `printmessage_1` that was referred to from the `rprintmsg` client program. The name of the output file for client stub routines is always formed in this way: if the name of the input file is `FOO.x`, the client stubs output file is called `FOO_clnt.c`.
- It created the server program that calls `printmessage_1` in `msg_proc.c`. This server program is named `msg_svc.c`. The rule for naming the server output file is similar to the previous one: for an input file called `FOO.x`, the output server file is named `FOO_svc.c`.

You are now ready to test the example. First, copy the server to a remote machine and run it. In this example, the machine is called `moon`.

```
moon% msg_server &
```

Server processes are run in the background, because they never exit. Then, on the local machine `sun`, print a message on `moon`'s console.

```
sun% rprintmsg moon "Hello, moon."
```

The message gets printed to `moon`'s console. You can print a message on anybody's console (including your own) with this program if you are able to copy the server to their machine and run it.

## Generating XDR Routines

The previous example only demonstrated the automatic generation of client and server RPC code. `rpcgen` may also be used to generate XDR routines, that is, the routines necessary to convert local data structures into network format and vice-versa.

This example presents a complete RPC service – a remote directory listing service, which uses `rpcgen` not only to generate stub routines, but also to generate the XDR routines.

## Protocol Description File

Here is the protocol description file:

```
/* dir.x: Remote directory listing protocol */

const MAXNAMELEN = 255; /* maximum length of a directory entry */

typedef string nametype<MAXNAMELEN>; /* a directory entry */
```

---

```

typedef struct namenode *namelist;      /* a link in the listing */

/* A node in the directory listing */

struct namenode
{
    nametype name          /* name of directory entry */
    namelist next;         /* next entry */
};

/* The result of a READDIR operation. */
union readdir_res switch (int errno)
{
    case 0:
        namelist list; /* no error : return directory listing */
    default:
        void;          /* error occurred: nothing else to return */
};

/* The directory program definition */

program DIRPROG
{
    version DIRVERS
    {
        readdir_res
        READDIR(nametype) = 1;
    } = 1;
} = 76;

```

Types (like `readdir_res` in this example) can be defined using the `struct`, `union`, and `enum` keywords, but those keywords should not be used in subsequent declarations of variables of those types. For example, if you define a union `foo`, you should declare using only `foo` and not `union foo`. `rpcgen` compiles RPC unions into C structures and it is an error to declare them using the `union` keyword.

## XDR Routines for Converting Data Types

Running `rpcgen` on `dir.x` creates four output files. Three are the same as before: header file, client stub routines, and server skeleton. The fourth file contains the XDR routines necessary for converting the data types declared into XDR format and vice-versa. These are output in the file `dir_xdr.c`.

### The READDIR Procedure

Here is the implementation of the `READDIR` procedure:

```

/* dir_proc.c: remote readdir implementation */

#include <rpc.h>
#include <dir.h>
#include "dir.h"

extern int errno;
extern char *malloc();
extern char *strdup();

readdir_res *
readdir_1(dirname)
    nametype *dirname;
{
    DIR *dirp;
    struct direct *d;

```

```

namelist nl;
namelist *nlp;
static readdir_res res;          /* must be static! */

/* Open directory */

dirp = opendir(*dirname);
if (dirp == NULL)
{
    res.errno = errno;
    return (&res);
}

/* Free previous result */

xdr_free(xdr_readdir_res, &res);

/* Collect directory entries. Memory allocated here will
 * be freed by xdr_free next time readdir_1 is called. */

nlp = &res.readdir_res_u.list;
while (d = readdir(dirp))
{
    nl = *nlp = (namenode *) malloc(sizeof(namenode));
    nl->name = strdup(d->d_name);
    nlp = &nl->next;
}
*nlp = NULL;

/* Return the result */

res.errno = 0;
closedir(dirp);
return (&res);
}

```

## The Client-side Program to Call the Server

Finally, there is the client-side program to call the server:

```

/* rls.c: Remote directory listing client */
#include <stdio.h>
#include <rpc.h>          /* always need this */
#include "dir.h"          /* will be generated by rpcgen */

extern int errno;

main(argc, argv)
    int argc;
    char *argv[];
{
    CLIENT *cl;
    char *server;
    char *dir;
    readdir_res *result;
    namelist nl;

    if (argc != 3)
    {

```



---

```

        fprintf(stderr, "usage: %s host directory\n", argv[0]);
        exit(1);
    }

    /* Remember what our command line arguments refer to */

    server = argv[1];
    dir = argv[2];

    /* Create client "handle" used for calling MESSAGEPROG on the
     * server designated on the command line. We tell the RPC
     * package to use the "tcp" protocol when contacting
     * the server. */

    cl = clnt_create(server, DIRPROG, DIRVERS, "tcp");
    if (cl == NULL)
    {

        /* Couldn't establish connection with server.
         * Print error message and die. */

        clnt_pcreateerror (server);
        exit(1);
    }

    /* Call the remote procedure readdir on the server */

    result = readdir_1(&dir, cl);
    if (result == NULL)
    {

        /* An error occurred while calling the server.
         * Print error message and die. */

        clnt_perror (cl, server);
        exit(1);
    }

    /* Okay, we successfully called the remote procedure. */

    if (result->errno != 0)
    {
        /* A remote system error occurred. Print error
         * message and die. */
        errno = result->errno;
        perror(dir);
        exit(1);
    }

    /* Successfully got a directory listing. Print it out. */

    for (nl = result->readdir_res_u.list;
         nl != NULL;
         nl = nl->next)
    {
        printf("%s\n", nl->name);
    }
    exit(0);
}

```

### Compiling and Running

Compile everything, and then run this routine:

```
sun% rpcgen dir.x
sun% cc rls.c dir_clnt.c dir_xdr.c -o rls
sun% cc dir_svc.c dir_proc.c dir_xdr.c -o dir_svc

sun% dir_svc &

moon% rls sun /usr/pub
.
.
.
ascii
eqnchar
greek
kbd
marg8
tabclr
tabs
tabs4
moon%
```

### Testing the Client and Server Procedures Together

A final note about rpcgen. The client program and the server procedure can be tested together as a single program by simply linking them with each other rather than with the client and server stubs. The procedure calls are executed as ordinary local procedure calls and the program can be debugged with a local debugger. When the program is working, the client program can be linked to the client stub produced by rpcgen and the server procedures can be linked to the server stub produced by rpcgen.

If you do this, you may want to comment out calls to RPC library routines, and have client-side routines call server routines directly.

## The C Preprocessor

The C preprocessor is run on all input files before they are compiled, so all the preprocessor directives are legal within an .x file.

### Symbols That May be Defined

These symbols may be defined, depending on which output file is getting generated:

Table 4-1 C Processor Definable Symbols	
Symbol	Description
RPC_HDR	For header-file output
RPC_XDR	For XDR routine output
RPC_SVC	For server-skeleton output
RPC_CLNT	For client stub output

---

## rpcgen Preprocessing

rpcgen does some preprocessing of its own. Any line that begins with a percent sign is passed directly into the output file without any interpretation of the line. This example demonstrates the preprocessing features:

```
/* time.x: Remote time protocol */

program TIMEPROG
{
    version TIMEVERS
    {
        unsigned int TIMEGET(void) = 1;
    } = 1;
} = 44;
#ifdef RPC_SVC
%int *
%timeget_1()
%{
%    static int thetime;
%
%    thetime = time(0);
%    return (&thetime);
%}
#endif
```

The % feature is not generally recommended, as there is no guarantee that the compiler puts the output where you intended.

## rpcgen Programming Notes

This section contains useful notes on rpcgen programming.

### Timeout Changes

RPC sets a default timeout of 25 seconds for RPC calls when `clnt_create` is used. This timeout may be changed using `clnt_control`. Here is a small code fragment to demonstrate use of `clnt_control`:

```
struct timeval tv;
CLIENT *cl;

cl = clnt_create("somehost", SOMEPROG, SOMEVERS, "tcp");
if (cl == NULL)
{
    exit(1);
}
tv.tv_sec = 60; /* change timeout to 1 minute */
tv.tv_usec = 0;
clnt_control(cl, CLSET_TIMEOUT, &tv);
```

### Handling Broadcast on the Server Side

When a procedure is known to be called via broadcast RPC, the server should reply only if it can provide some useful information to the client.

To prevent the server from replying, a remote procedure can return NULL as its result, and the server code generated by rpcgen detects this and does not send out a reply.

Here is an example of a procedure that replies only if it thinks it is an NFS server:

```
void *
reply_if_nfsserver()
{
    char notnull;          /* just here so we can use its address */

    if (access("/etc/exports", F_OK) < 0)
    {
        return (NULL);      /* prevent RPC from replying */
    }

    /* Return non-null pointer so RPC will send out a reply */

    return ((void *)&notnull);
}
```

For example, if the procedure returns type void \*, it must return a non-null pointer if it wants RPC to reply for it.

## Other Information Passed to Server Procedures

Server procedures often want to know more about an RPC call than just its arguments.

Getting authentication information is important to procedures that want to implement some level of security.

This extra information is actually supplied to the server procedure as a second argument. Here is an example to demonstrate its use. Rewrite the previous `printmessage_1` procedure to only let root users print a message to the console:

```
int *
printmessage_1(msg, rq)
    char **msg;
    struct svc_req *rq;
{
    static int result;      /* Must be static */
    FILE *f;
    struct suthunix_parms *aup;

    aup = (struct authunix_parms *)rq->rq_clntcred;
    if (aup->aup_uid != 0)
    {
        result = 0;
        return (&result);
    }
    .
    .   Same code as before
    .
}
```

## The RPC Language

The RPC language is an extension of XDR language. The sole extension is the addition of the program type. For a complete description of the XDR language syntax, read the XDR Protocol specification, RFC 1014. For a description of the RPC extensions to the XDR language, read RFC 1057.

The XDR language is very close to C; you know C, you know most of XDR. The following topics describe the syntax of the RPC language, showing a few examples, and showing how the various RPC and XDR type definitions get compiled into C type definitions in the output header file.

---

## Definitions

An RPC language file consists of a series of definitions:

```
definition-list:
    definition ";"
    definition ";" definition-list
```

It recognizes these types of definitions:

```
definition:
    enum-definition
    struct-definition
    union-definition
    typedef-definition
    const-definition
    program-definition
```

## Structures

An XDR struct is declared almost exactly like its C counterpart:

```
struct-definition:
    "struct" struct-ident "{"
        declaration-list
    "}"

declaration-list:
    declaration ";"
    declaration ";" declaration-list
```

Here is an XDR structure to a two-dimensional coordinate, and the C structure that gets compiled into the output header file:

```
struct coord {
    int x;
    int y;
};
```

becomes

```
struct coord {
    int x;
    int y;
};
typedef struct coord coord;
```

The output is identical to the input, except for the added typedef at the end of the output. This lets you use coord instead of struct coord when declaring items.

## Unions

XDR unions are discriminated unions, and look quite different from C unions. They are more analogous to Pascal variant records than they are to C unions:

```
union-definition:
    "union" union-ident "switch" "(" declaration ")" "{"
        case-list
    "}"

case-list:
    "case" value ":" declaration ";"
```

```
"default" ":" declaration ";"  
"case" value ":" declaration ";" case-list
```

Here is an example of a type that might be returned as the result of a read data operation. If there is no error, return a block of data. Otherwise, do not return anything:

```
union read_result switch (int errno)  
{  
    case 0:  
        opaque data[1024];  
    default:  
        void;  
};
```

... gets compiled into this code:

```
struct read_result  
{  
    int errno;  
    union  
    {  
        char data[1024];  
    } read_result_u;  
};  
typedef struct read_result read_result;
```

The union component of the output struct has the same name as the type, except for the trailing `_u`.

## Enumerations

XDR enumerations have the same syntax as C enumerations:

```
enum-definition:  
    "enum" enum-ident "{"  
    enum-value-list  
    "}"  
  
enum-value-list:  
    enum-value  
    enum-value "," enum-value-list  
  
enum-value:  
    enum-value-ident  
    enum-value-ident "=" value
```

Here is a short example of an XDR enum, and the C enum that it gets compiled into:

```
enum colortype {  
    RED = 0,  
    GREEN = 1,  
    BLUE = 2  
};
```

...gets compiled into:

```
enum colortype {  
    RED = 0,  
    GREEN = 1,  
    BLUE = 2  
};  
typedef enum colortype colortype;
```

---

## Typedefs

XDR typedefs have the same syntax as C typedefs:

```
typedef-definition:
    "typedef" declaration
```

Here is an example that defines a `fname_type` used for declaring file name strings that have a maximum length of 255 characters:

```
typedef string fname_type<255>;
becomes
typedef char *fname_type;
```

## Constants

XDR contains symbolic constants that may be used wherever an integer constant is used; for example, in array size specifications:

```
const-definition:
    "const" const-ident "=" integer
```

This example shows how to define the constant `DOZEN` equal to 12:

```
const DOZEN = 12;
becomes
#define DOZEN 12
```

## Programs

RPC programs are declared using this syntax:

```
program-definition:
    "program" program-ident "{"
        version-list
    "}" "=" value
version-list:
    version ";"
    version ";" version-list
version:
    "version" version-ident "{"
        procedure-list
    "}" "=" value
procedure-list:
    procedure ";"
    procedure ";" procedure-list
procedure:
    type-ident procedure-ident "(" type-ident ")" "=" value
```

Here is the time protocol, revisited:

```
/* time.x: Get or set the time. Time is represented
 * as number of seconds since 0:00, January 1, 1970. */

program TIMEPROG
{
    version TIMEVERS
    {
```

```
        unsigned int TIMEGET(void) = 1;
        void TIMESET(unsigned) = 2;
    } = 1;
} = 44;
```

This file compiles into #defines in the output header file:

```
#define TIMEPROG 44
#define TIMEEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

## Declarations

XDR supports these kinds of declarations:

- Simple-declaration
- Fixed-array-declaration
- Variable-array-declaration
- Pointer-declaration

### Simple Declarations

These are just like simple C declarations:

```
simple-declaration:
    type-ident variable-ident
```

For example,

```
    colortype color;
becomes
    colortype color;
```

### Fixed-length Array Declarations

These are just like C array declarations:

```
fixed-array-declaration:
    type-ident variable-ident "[" value "]"
```

For example,

```
    colortype palette[8];
becomes
    colortype palette[8];
```

### Variable-Length Array Declarations

These have no explicit syntax in C. The XDR format uses angle-brackets:

```
variable-array-declaration:
    type-ident variable-ident "<" value ">"
    type-ident variable-ident "<" ">"
```



---

The maximum size is specified between the angle brackets. The size may be omitted, which indicates that the array may be of any size:

```
int heights<12>;           /* at most 12 items */
int widths<>;              /* any number of items */
```

Since variable-length arrays have no explicit syntax in C, these declarations are actually compiled into structs.

For example, the heights declaration gets compiled into this struct:

```
struct
{
    u_int heights_len;      /* # of items in array */
    int *heights_val;       /* pointer to array */
} heights;
```

The number of items in the array is stored in the `_len` component and the pointer to the array is stored in the `_val` component. The first part of each component name is the same as the name of the declared XDR variable.

## Pointer Declarations

These are made in XDR exactly as they are in C. You cannot send pointers over the network, but you can use XDR pointers for sending recursive data types such as lists and trees. The type is actually called “optional-data” in XDR language, not “pointer”:

```
pointer-declaration:
    type-ident "*" variable-ident
```

For example,

```
listitem *next; --> listitem *next;
```

## Special Cases

There are a few exceptions to the previously described rules.

### Booleans

C has no built-in boolean type. However, the RPC library does have a boolean type called `bool_t` that is either `TRUE` or `FALSE`. Things declared as type `bool` in XDR language are compiled into `bool_t` in the output header file.

```
bool married;
becomes
bool_t married;
```

### Strings

C has no built-in string type, but instead uses the null-terminated `char *` convention. In XDR language, strings are declared using the `string` keyword and compiled into `char *`s in the output header file. The maximum size contained in the angle brackets specifies the maximum number of characters allowed in the strings (not counting the `NULL` character). The maximum size may be left off, which indicates a string of arbitrary length.

Here are two examples:

```
string name<32>;
```

becomes

```
char *name;
```

The second example:

```
string longname<>;
```

becomes

```
char *longname;
```

### Opaque Data

Opaque data is used in RPC and XDR to describe untyped data, that is, just sequences of arbitrary bytes. It may be declared either as a fixed or variable length array.

Here are two examples:

```
opaque diskblock[512];
```

becomes

```
char diskblock[512];
```

The second example is:

```
opaque filedata<1024>;
```

becomes

```
struct
{
    u_int filedata_len;
    char *filedata_val;
} filedata;
```

### Voids

In a void declaration, the variable is not named. The declaration is just void and nothing else. Void declarations can only occur in two places: union definitions and program definitions as the argument or result of a remote procedure.