



SAS Publishing



# The PROTO Procedure

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2003. *The PROTO Procedure*. Cary, NC: SAS Institute Inc.

### **The PROTO Procedure**

Copyright © 2003, SAS Institute Inc., Cary, NC, USA

All rights reserved. Produced in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

**U.S. Government Restricted Rights Notice:** Use, duplication, or disclosure of this software and related documentation by the U.S. government is subject to the Agreement with SAS Institute and the restrictions set forth in FAR 52.227-19, Commercial Computer Software-Restricted Rights (June 1987).

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, September 2003

SAS Publishing provides a complete selection of books and electronic products to help customers use SAS software to its fullest potential. For more information about our e-books, e-learning products, CDs, and hard-copy books, visit the SAS Publishing Web site at [support.sas.com/pubs](http://support.sas.com/pubs) or call 1-800-727-3228.

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

# Chapter 7

## The PROTO Procedure

### Chapter Contents

---

<b>INTRODUCTION</b> . . . . .	669
<b>SYNTAX</b> . . . . .	669
PROC PROTO Statement . . . . .	670
MAPMISS Statement . . . . .	671
LINK Statement . . . . .	671
Function Prototypes . . . . .	672
<b>C LANGUAGE TYPES IN SAS</b> . . . . .	674
Basic C Language Types . . . . .	674
C Structures in SAS . . . . .	677
Enumerations in SAS . . . . .	679
C-Source in SAS . . . . .	680
C-Language Limitations in SAS . . . . .	682
Helper Functions . . . . .	683
<b>EXAMPLE</b> . . . . .	687
Splitter Function Example . . . . .	687
<b>SUBJECT INDEX</b> . . . . .	689
<b>SYNTAX INDEX</b> . . . . .	691



# Chapter 7

## The PROTO Procedure

---

### Introduction

One of the key components of a good enterprise-wide risk management system is the capability to easily integrate external functions into the system. Usually, these external functions used to transform financial market data and to price financial instruments. They are necessary in most market and credit risk management tasks and are usually developed in-house by financial analysts or supplied by pricing-function vendors.

The PROTO procedure enables you to register, in batch, external functions written in the C or C++ programming languages for use in SAS. The functions are then available for use in method programs, which are written in [COMPILE procedure](#) batch code or in the Method Program Editor (Chapter 5, *SAS Risk Dimensions 4.1: Graphical User Interface Reference*) of the Risk Dimensions GUI. You can also test your function packages by using the [COMPILE procedure](#).

---

### Syntax

The following statements are used in the PROTO procedure:

**PROC PROTO PACKAGE=** *catalog-entry options*;

**MAPMISS** *type1=value1 type2=value2 ...*;

**LINK** *load-module*;

**Declarations and Preprocessor Statements**

---

## PROC PROTO Statement

**PROC PROTO PACKAGE=** *catalog-entry options*;

**PACKAGE=** *catalog-entry*

identifies the SAS catalog entry where the function package information is saved. The catalog-entry is a three-level catalog name of the form: *member* or *library.catalog.member*. This parameter is required.

**LABEL=** *package-label*

specifies a text string that describes or labels the package. The maximum length of the label is 256 characters.

**STRUCTPACK<sub>n</sub> or PACK<sub>n</sub>**

specifies that all structures in this package were compiled with the given N-BYTE packing pragma. That is, 'STRUCTPACK4' specifies that all structures in the package were compiled with the '#pragma pack(4)' option. This option is only valid on a 32-bit PC operating environment; otherwise, it is ignored.

**STDCALL**

indicates that all functions in the package are to be called using the '\_\_\_stdcall' convention. This option is only valid on a 32-bit PC operating environment; otherwise, it is ignored.

---

## MAPMISS Statement

```
MAPMISS POINTER=NULL
        INT=integer-value
        DOUBLE=double-value
        LONG=long-value
        SHORT=short-value;
```

The **MAPMISS** statement is used to specify alternative values, by type, to pass functions if values are missing. You can specify alternatives for any or all of the five types shown in the **MAPMISS** statement. If you set **POINTER=NULL**, a **NULL** valued pointer is passed to the functions for pointer variables that are missing. If you do not specify a mapping for a type that is used as arguments to a function, the function is not called when an argument of that type is missing.

---

## LINK Statement

**LINK** *load-module*;

The **LINK** statement specifies the name and, optionally, the path of the load module that contains your functions. You can add more **LINK** statements to include as many libraries as needed for your prototypes. Full path names can be specified (as appropriate for your operating environment):

```
link 'c:\mylibs\zzz.dll';
link 'c:\mylibs\zzz';
link '/users/me/mylibs/zzz';
```

Full path name specification is the safest and recommended way to link your modules when using the **PROTO** procedure.

If the full path-name is not specified, the module must be located in the current **SYSTEM** path or **SAS TKEXTENSION PATH**. This method of finding your module is much more ambiguous and not recommended:

```
link 'zzz.dll';
link 'mylib';
```

Note that, in the preceding examples, your module's extension need not be specified, but SAS will attempt to load your module with the appropriate extension for your operating environment.

All functions of interest must be externally declared in your load module so that SAS can find them. For most platforms, this is the default behavior for the compiler. However, many PC compilers don't export function names, by default. For most PC compilers the following examples demonstrate how to declare your functions for external loading:

```
__declspec( dllexport ) int myfunc( int, double );
__declspec( dllexport ) int price2( int a, double zzz );
```

---

## Function Prototypes

Function prototypes are registered (declared) in the PROTO procedure in the following general form:

```
return-type function-name(arg-type <arg-name>
    / <iotype> <arg-label>,...) <options>;
```

The following items must be specified in the PROTOTYPE statement:

### return-type

specifies a C Language type for the returned value. See “Supported C Return Types” for a list of supported C types. The return type can be preceded by either the **unsigned** or **Exceldate** modifiers. Exceldate is needed if the return type is a Microsoft Excel date. (Microsoft Excel is a registered trademark.)

### function-name

specifies the name of the function to be registered. Function names within a given package must be unique in the first 32 characters. Functions names do not need to be unique across different packages.

### arg-type

specifies the C language type for the function argument. You must specify the type for each argument in the function’s argument list. The argument list must be enclosed in parenthesis. For a list of supported C types, see “Supported C Argument Types”. The argument type can be preceded by either the **unsigned**, **const**, or **Exceldate** modifiers. Exceldate is needed if the argument type is a Microsoft Excel date.

The following options can be specified for each argument:

### arg-name

specifies the name of the argument.

### iotype

specifies the I/O type of the argument (**I** for input, **O** for output, or **U** for update).

### arg-label

specifies a description or label for the argument.

The following options can be specified per each function and are used for documentation and identification only:

### LABEL=“text-string”

specifies a description or a label for the function. The text string must be enclosed in quotation marks.

### KIND | GROUP= "group-type"

specifies the group that the function belongs to. The KIND= or GROUP= option allows for convenient grouping of functions in a package. Any quoted string (up to a maximum 40 characters) can be used to group similar functions. The special *group-types* provided for Risk Dimensions that do not require quotation marks are: INPUT (Instrument Input), TRANS (Risk Factor Transformation), PRICING (Instrument Pricing), and PROJECT. The default is PRICING.

### Supported C Return Types

The following C return types are supported in the PROTO procedure:

**Table 7.1.** Supported Return Types

Function Prototype	SAS Variable Type	C Variable Type
short	Numeric	short, short *, short **
short *	Numeric, Array	short, short *, short **
int	Numeric	int, int *, int **
int *	Numeric, Array	int, int *, int **
long	Numeric	long, long *, long **
long *	Numeric, Array	long, long *, long **
double	Numeric	double, double *, double **
double *	Numeric, Array	double, double *, double **
char *	Character	char *, char **
struct *	struct	struct *, struct **
void		void

### Supported C Argument Types

The following C argument types are supported in the PROTO procedure:

**Table 7.2.** Supported Types for External C arguments

Function Prototype	SAS Variable Type	C Variable Type
short	Numeric	short, short *, short **
short *	Numeric, Array	short, short *, short **
short **	Array	short *, short **
int	Numeric	int, int *, int **
int *	Numeric, Array	int, int *, int **
int **	Array	int *, int **
long	Numeric	long, long *, long **
long *	Numeric, Array	long, long *, long **
long **	Array	long *, long **
double	Numeric	double, double *, double **
double *	Numeric, Array	double, double *, double **
double **	Array	double *, double **
char *	Character	char *, char **
char **	Character	char *, char **
struct *	Structure	struct *, struct **
struct **	Structure	struct *, struct **

---

## C Language Types in SAS

---

### Basic C Language Types

The SAS language supports only two basic types; character and numeric. These two types correspond to an array of characters and a *double* in the C programming language. When SAS variables are used as arguments to external C functions, they are converted (cast) into the proper C types. This section contains details about how the conversions are made and what conversions are allowed.

Character variables can be used for an argument that requires a "char \*" value only. The passed character string will be a NULL-terminated at the allocated length of the string. If the current length of the character string is less than the allocated length, the character string is blank padded at the end:

```
length str $ 10;
str = 'hello';
call foo( str );
```

In this example, the allocated length of `str` is 10, but the current length is 5. Thus when the string is NULL-terminated at the allocated length, 'hello ' is passed to the function `foo`.

To avoid this blank padding, use the SAS function `TRIM()` on the parameter within the function call:

```
length str $ 10;
str = 'hello';
call foo( TRIM(str) );
```

In this case, 'hello' is passed to function `foo`.

Functions that return "char \*" can return a NULL or a zero delimited string that is copied to the SAS variable.

Numeric variables can be used for an argument that requires a short, an *int*, a *long*, or a *double* type, and pointers to those types. The numeric variable is converted to the required type automatically. If the conversion fails, the function is not called and the outputs to the function are set to missing. If pointers to these types are requested, the address of the converted value is passed. On return from the call, the value is converted back to a double and stored in the SAS variable. SAS scalar variables cannot be passed as arguments that require two or more levels of indirections. For example, a SAS variable cannot be passed as an argument that requires a cast to a *long \*\** type.

SAS variables that contain missing values are converted according to how the function that is being called has mapped missing values when using the PROTO procedure. All returned variables from the function are checked for the mapped missing

values and converted to SAS missing values. For example, if an argument to a function is missing, and it was to be converted to an integer and an integer was mapped to -99, then -99 is passed to the function. If the same function returns an integer with the value -99, then the variable that this value is returned to would have a value of missing.

To make it easier to interface to external C functions, the COMPILE procedure has been enhanced to support most of these C types. As Table 7.3 shows, there is no way to return and save a pointer to any type in a SAS variable. Pointers are always dereferenced, and their contents are converted and copied to SAS variables. The EXTERNC statement is used to specify C variables in PROC COMPILE statements. The syntax of the EXTERNC statement is

```
EXTERNC DOUBLE | INT | LONG | SHORT | CHAR [*[*]] var1, [*[*]] var2...,
[*[*]] varn;
```

Refer to Table 7.3 to see how these variables are treated when they are on the left-side of an expression. Refer to Table 7.4 to see how these variables are treated when they are passed as arguments to an external C function. Table 7.3 shows the automatic casting that is performed for a short type on the right-side of an assignment. This table lists all the combinations of short types with SAS variables that are permitted. The table for *int*, *long*, and *double* types can be intuited by substituting any of the types for *short* in this table. If any of the pointers are null and require dereferencing, then the result is set to missing if there is a missing value set for the result variable (see the MAPMISS statement).

**Table 7.3.** Automatic Type Casting for the *short* Type in an Assignment Statement

LHS Type	RHS Type	Cast Performed
short	SAS Numeric	y = (short) x
short	short	y = x
short	short *	y = * x
short	short **	y = ** x
short *	SAS Numeric	not supported
short *	short	y = & x
short *	short *	y = x
short *	short **	y = * x
short **	SAS Numeric	not supported
short **	short *	y = & x
short **	short **	y = x
SAS Numeric	short	y = (double) x
SAS Numeric	short *	y = (double) * x
SAS Numeric	short **	y = (double) ** x

**Table 7.4.** Allowed types for External C arguments

Function Prototype	SAS Variable Type	C Variable Type
short	Numeric	short, short *, short **
short *	Numeric, Array	short, short *, short **
short **	Array	short *, short **
int	Numeric	int, int *, int **
int *	Numeric, Array	int, int *, int **
int **	Array	int *, int **
long	Numeric	long, long *, long **
long *	Numeric, Array	long, long *, long **
long **	Array	long *, long **
double	Numeric	double, double *, double **
double *	Numeric, Array	double, double *, double **
double **	Array	double *, double **
char *	Character	char *, char **
char **	Character	char *, char **
struct *	Structure	struct *, struct **
struct **	Structure	struct *, struct **

**Note:** Automatic conversion between 2 different C types is never performed.

---

## C Structures in SAS

Many C language libraries contain functions that have structure pointers as arguments. In SAS, structures can only be defined in PROC PROTO. After being defined in PROC PROTO, they can be declared or instantiated by using the COMPILE procedure.

A C structure is a template that is applied to a contiguous piece of memory. Each entry in the template is given a name and a type. The type of each element determines the number of bytes that are associated with each entry and how each entry is to be used. Because of various alignment rules and base-type sizes, SAS software relies on the current machine compiler to determine the location of each entry in the memory of the structure.

### *Declaring and Referencing Structures in SAS*

The syntax of a structure declaration in SAS is the same as for C non-pointer structure declarations.

```
struct structure_name structure_instance;
```

Each structure is set to zero at declaration time, and retains the value from the previous pass through the data to start the next pass.

Structure elements are referenced by using the static period (.) notation of C. There is no pointer syntax for SAS. If a structure points to another structure, the only way to reference the structure that is pointed to is by assigning the pointer to a declared structure of the same type, and using that declared structure to access the elements.

If a structure entry is a *short*, an *int*, or a *long* type and it is referenced in an expression, it is first cast to a *double* type, then used in the calculations. If a structure entry is a pointer to a base type, the pointer is dereferenced and the value is returned. If the pointer is NULL, a missing value is returned. The missing value assignments that are made in the PROC PROTO code are used when conversions fail or when missing values are assigned to non-double structure entries.

**Structure Example**

```
proc proto package=sasuser.mylib.struct
  label="package of structures";

  #define MAX_IN 20;

  struct ZZZ {
    double dval;
    int ival;
    char * buffer;
    long * lptr;
    struct {
      int  a;
      short b;
    } inner;
  };

  struct ZZZ2 {
    struct ZZZ * tom;
  };

run;

proc compile library=sasuser.mylib;
  struct ZZZ result;
  struct ZZZ2 result2;
  externc long lval;
  lval = 23;
  result.dval = 2.3;
  result.ival = 2;
  result.buffer = "hello";
  result.lptr = lval;
  result.inner.a = 1;
  result.inner.b = 1;
  put result=;
  result2.tom = result;
  put result2=;

run;
```

## Enumerations in SAS

Enumerations are mnemonics for integer numbers. Enumerations enable you to set a literal name as a specific number and aid in the readability and supportability of C programs. Enumerations are used in C language libraries to simplify the return codes and to make changes to the associated value invisible in future releases. After a C program is compiled, enumeration names are no longer accessible.

### Enumerated Types Example

This example demonstrates setting up two enumerated value types in PROC PROTO: **YesNoMaybeType** and **Tens**. Both types are referenced in the structure **EStructure**.

```
proc proto package=sasuser.mylib.str2
  label="package of structures";

  typedef enum
  {
    True, False, Maybe
  } YesNoMaybeType ;

  typedef enum {
    Ten=10, Twenty=20, Thirty=30, Forty=40, Fifty=50
  } Tens;

  typedef struct {
    short          rows ;
    short          cols ;
    YesNoMaybeType type;
    Tens           dollar;
    ExerciseArray  dates;
  } EStructure ;
run;
```

The following PROC COMPILE code demonstrates how to access these enumerated types.

```
proc compile inlib=sasuser.mylib;
  EStructure mystruct;

  mystruct.type = &True;
  mystruct.dollar = &Twenty;
run;
```

In the preceding example, the enumerated values set up in PROC PROTO are actually implemented in SAS as macro variables. Therefore, they must be accessed by using the ampersand & symbol.

---

## C-Source in SAS

PROC PROTO can be used in a limited fashion to compile your external C functions. The C source code can be specified in PROC PROTO as follows:

```
EXTERNC function-name;
```

```
C-source-statements
```

```
EXTERNCEND ;
```

The *function-name* tells PROC PROTO what function's source code is specified between the EXTERNC statement and the EXTERNCEND statement. When PROC PROTO compiles your source code, it will include any structure definitions and C function prototypes that are currently declared. However, typedefs and defines are not included.

This functionality is provided to allow creation of simple 'helper' functions that facilitate the interface to pre-existing external C libraries. Any valid C statements are permitted except the #include statement. Only a limited subset of the C-stdlib functions are available. However, you can call any other C function that is already declared within the current PROC PROTO step.

The following C-stdlib functions are available:

**Table 7.5.** Supported stdlib functions

double <b>sin</b> ( double x )	Returns the sine of x (radians)
double <b>cos</b> ( double x )	Returns the cosine of x (radians)
double <b>tan</b> ( double x )	Returns the tangent of x (radians)
double <b>asin</b> ( double x )	Returns the arcsine of x (-pi/2 to pi/2 radians)
double <b>acos</b> ( double x )	Returns the arccosine of x (0 to pi radians)
double <b>atan</b> ( double x )	Returns the arctangent of x (-pi/2 to pi/2 radians)
double <b>atan2</b> ( double x, double y )	Returns the arctangent of y/x (-pi to pi radians)
double <b>sinh</b> ( double x )	Returns the hyperbolic sine of x (radians)
double <b>cosh</b> ( double x )	Returns the hyperbolic cosine of x (radians)
double <b>tanh</b> ( double x )	Returns the hyperbolic tangent of x (radians)
double <b>exp</b> ( double x )	Returns the exponential value of x
double <b>log</b> ( double x )	Returns the logarithm of x
double <b>log2</b> ( double x )	Returns the logarithm of x base-2
double <b>log10</b> ( double x )	Returns the logarithm of x base-10
double <b>pow</b> ( double x, double y )	Returns x raised to the y power $x^y$
double <b>sqrt</b> ( double x )	Returns the square root of x
double <b>ceil</b> ( double x )	Returns the smallest integer not less than x
double <b>fmod</b> (double x, double y)	Returns the remainder of (x/y)
double <b>floor</b> ( double x )	Returns the largest integer not greater than x
int <b>abs</b> (int x)	Returns the absolute value of x
double <b>fabs</b> ((double)	Returns the absolute value of x
int <b>min</b> (int x, int y)	Returns the minimum of x and y
double <b>fmin</b> (double x, double y)	Returns the minimum of x and y
int <b>max</b> (int x, int y)	Returns the maximum of x and y
double <b>fmax</b> (double x, double y)	Returns the maximum of x and y
char * <b>malloc</b> ( int x )	Allocate memory of size x
void <b>free</b> ( char *)	Free memory allocated with malloc

Here is an example of a simple C function written directly in PROC PROTO:

```
proc proto package=sasuser.mylib.foo;
  struct MYSTRUCT {
    short a;
    long b;
  };
  int fillMyStruct( short a, short b, struct MYSTRUCT * s );
  externc fillMyStruct;
  int fillMyStruct( short a, short b, struct MYSTRUCT * s ) {
    s->a = a;
    s->b = b;
    return( 0 );
  }
  externcend;
run;
```

---

## C-Language Limitations in SAS

The limitations for the C language specifications in the PROTO procedure are as follows.

- #defines must be followed by a semicolon (;) and be numeric in value.
- The #define functionality is limited to simple replacement and unnested expressions.
- C preprocessor statements #include and #if are not supported. The SAS macro %INC can be used in place of #include.
- A maximum of two levels of indirection are allowed for structure elements. So elements like *double \*\*\** are not allowed. If these element types are needed in the structure but are not accessed on the SAS side, placeholders can be used.
- The float type is not supported.
- The union type is not supported. However, if you only plan to use one element of the union, declare the variable for the union as the type for that element.
- All non-pointer references to other structures must be defined before they are used.
- The **enum** keyword does not work in a structure. In order to specify an **enum** in a structure, use a **typedef**.
- Structure elements that have the same alphanumeric name, but are written in different cases (for example, ALPHA, Alpha, alpha, and so on) are not supported. SAS is not case sensitive; therefore, all structure elements must be unique when compared in a case-insensitive program.

---

## Helper Functions

Several helper functions have been added to extend the SAS language to handle C language constructs that do not fit naturally into the SAS language.

### *ISNULL C Helper Function*

ISNULL determines whether or not a pointer element of a structure is NULL. When a pointer-valued variable (a structure entry or the structure itself) is specified, the ISNULL function returns 1.0 if its argument is NULL and returns 0.0 otherwise.

```

/* loop over linked list */
do while ( ^isnull( l ) );
  put l.num;
  l = l.next;
end;

```

**double ISNULL ( *pointer-element* );**

For example, the LINKLIST structure and function **get\_list** are defined when using PROC PROTO as shown below. The **get\_list** function is an external C routine that generates a linked list with as many elements as requested.

```

struct LINKLIST {
  double value;
  struct LINKLIST * next;
};

struct LINKLIST * get_list( int );

```

The following COMPILE code segment demonstrates using the ISNULL() helper function to loop over the linked list that is created by get\_list() and print out the elements:

```

struct LINKLIST list;

list = get_list( 3 );
put list.value=;

do while ( ^ISNULL(list.next ) );
  list = list.next;
  put list.value=;
end;

```

The following output is produced by this code.

The SAS System  
COMPILE Procedure

```
LIST.value=0  
LIST.value=1  
LIST.value=2
```

### **SETNULL C Helper Function**

SETNULL sets a pointer element of a structure to NULL. When a pointer-valued variable (a structure entry) is specified, the SETNULL subroutine sets the pointer to NULL.

```
call setnull( l2.next );
```

**call SETNULL ( *pointer-element* );**

Assuming that the same LINKLIST structure that is described above is defined using by PROC PROTO, the SETNULL() subroutine could be used to set the next element to NULL:

```
struct LINKLIST list;  
call SETNULL(list.next);
```

### STRUCTINDEX C Helper Function

STRUCTINDEX enables access to each structure element in an array of structures.

When a structure contains an array of structures, each structure element of the array can be accessed by using the STRUCTINDEX() subroutine.

**call STRUCTINDEX ( *struct\_array*, *index*, *struct\_element* );**

The *index* parameter is a 1-based index as is used in SAS arrays. An example is shown in the following code, which assumes that the structures and function are defined by using PROC PROTO.

```

struct POINT {
    short s;
    int i;
    long l;
    double d;
};

struct POINT_ARRAY {
    int length;
    struct POINT * p;
    char name[32];
};

struct POINT * struct_array( int );

```

The following PROC COMPILE code segment demonstrates using the STRUCTINDEX() helper function to get and to set each POINT structure element of an array p in the 'POINT\_ARRAY' structure:

```

struct POINT_ARRAY pntarray;
struct POINT pnt;

/* call struct_array() to allocate array of 2 POINT structs */
pntarray.p = struct_array( 2 );
pntarray.plen = 2;
pntarray.name = "My funny structure";

/* get each element using the STRUCTINDEX() call and set values */
do i = 1 to 2;
    call STRUCTINDEX( pntarray.p, i, pnt );
    put "Before setting the" i "element: " pnt=;
    pnt.s = 1;
    pnt.i = 2;
    pnt.l = 3;
    pnt.d = 4.5;
    put "After setting the" i "element: " pnt=;
end;

```

The output produced by the preceding code is:

The SAS System  
COMPILE Procedure

```
Before setting the 1 element: PNT {s=0, i=0, l=0, d=0}  
After setting the 1 element: PNT {s=1, i=2, l=3, d=4.}  
Before setting the 2 element: PNT {s=0, i=0, l=0, d=0}  
After setting the 2 element: PNT {s=1, i=2, l=3, d=4.5}
```

---

## Example

---

### Splitter Function Example

---

```
proc proto package=sasuser.myfuncs.mathfun
    label="package of math functions"
    link "/users/davis/com/mathfun";
    link "/local/libs/math";
    int split(int x "number to split")
        label="splitter function" kind=PRICING;
    int cashflow(double amt, double rate, int periods,
        double * flows / iotype=0)
        label="cash flow function" kind=PRICING;
run;

proc compile libname=sasuser.myfuncs;
    array flows[20];
    a=split(32);
    put a;
    b=cashflow(1000, .07, 20, flows);
    put b;
    put flows;
    quit;
run;
```



# Subject Index

## B

batch processing  
function library packages, [669](#)

## C

C language  
C-Source in SAS, [680](#)  
enumerations in SAS, [679](#)  
helper functions, PROTO procedure, [683](#)  
structures in SAS, [677](#)

## F

function library packages, [669](#)

## P

PROTO procedure  
C language helper functions, [683](#)  
declaring structures in SAS, [677](#)  
example, [687](#)  
general information, [669](#)  
limitations, [682](#)  
referencing structures in SAS, [677](#)  
supported C argument types, [673](#)  
supported C return types, [673](#)

## S

structures in SAS  
PROTO procedure, [677](#)



# Syntax Index

## I

ISNULL C helper function  
PROTO procedure, [683](#)

## L

LINK statement  
PROTO procedure, [671](#)

## M

MAPMISS statement  
PROTO procedure, [671](#)

## P

PROC PROTO statement  
PROTO procedure, [670](#)  
PROTO procedure, [669](#)  
ISNULL C helper function, [683](#)  
LINK statement, [671](#)  
MAPMISS statement, [671](#)  
PROC PROTO statement, [670](#)  
PROTOTYPE statement, [672](#)  
SETNULL C helper function, [684](#)  
STRUCTINDEX C helper function, [685](#)  
syntax, [669](#)  
PROTOTYPE statement  
PROTO procedure, [672](#)

## S

SETNULL C helper function  
PROTO procedure, [684](#)  
STRUCTINDEX C helper function  
PROTO procedure, [685](#)