parc
A Xerox Company

Computing Science Laboratory

PARC Library

Canonical C Functions

Glenn Scott[1]

Abstract
This is an implementation guide for canonical C functions to enforce consistent use across the system. The uniform naming of specific functions with behavior results in quicker understanding for developers and better interoperability between modules.

Keywords
Design Guide — C Language — Software Quality

[1] *Computing Science Laboratory, Palo Alto Research Center*
*__Corresponding author__: glenn.scott@parc.com

Contents

1. Function Names and Semantics

A consistent function naming scheme:

- leads to a better understanding of exactly what is meant,
- reduces the size of the vocabulary necessary to understand a collection of functions by their names, d
- permits functions to be grouped and recognized at sight, and
- makes the composition of new function names more straightforward.

See the document titled "Naming Conventions in C Programs" for a description of the naming conventions to be used for namespaces, modules, functions, and constants.

Following is an enumeration of canonical function names and their description. Each may appear at most once in a namespace. Any function with one of the following names must implement the behavior described in a way that is appropriate for that namespace.

2. Basic

2.1 Create

Functions with the word Create in their name create references to things (memory, files, other resources) that the caller must manage. For example, a common usage it to create a type, like `parcLinkedList_Create` which creates an instance of `PARCLinkedList` and the reference returned by the function must be ultimately released via `parcLinkedList_Release`.

Specifically, the reverse is also true, every function that creates a reference to a thing that the caller must manage must use the word `Create` according to these guidelines.

If a type has multiple ways to create an instance, the create functions (that are in addition to the default), are named `CreateFrom`*Something*. For example, `parcBuffer_CreateFromArray`, creates a `PARCBuffer` with the contents of a given C array.

In other cases, functions create references to other types derived from an original. For example, `parcLinkedList_CreateIterator` creates an instance of `PARCIterator` that will iterate through a `PARCLinkedList`. Note that `parcLinkedList_CreateFromIterator` would have a different and rather obvious meaning.

```
ReturnType nameSpace_Create...(
        parameters
)
```

This function creates a value from the given input parameters. The returned value must be managed by the caller. For example, functions that return a pointer to a reference counted

structure require the caller to ensure that the reference is decremented when the life of the structure ends.

Typically, this function creates the most basic or primitive value for the type. Other creation functions may create instances of the type from other forms.

See also: `_CreateFrom`

2.2 CreateFrom

```
ReturnType nameSpace_CreateFrom...(
    Type    from
)
```

This function creates a value from an alternative representation of the value (e.g. a string, a JSON object, etc.).

Like all Create functions, the return value must be managed by the caller. This is similar to the **_Create** function.

2.3 To

```
ReturnType nameSpace_To...(
    Type    original
)
```

This function returns a new value as a representation of the original input value.

For example, a function named after the form `parcBuffer_ToString`((PARCBuffer *buffer)), returns a pointer to a null-terminated C string containing a representation of `PARCBuffer`. The caller is responsible for managing the returned value. For example, if the returned value is a PARC reference counted object, it must be released via its corresponding `Release` function.

2.4 Get

```
ReturnType nameSpace_Get...(
        parameters
)
```

A `Get` function returns a value or reference to a value in the corresponding instance. A new reference is not acquired. The caller must acquire its own reference to the value if it must exist beyond the life of the source instance.

A `Get` function should not be used to get a computed or derived value. Instead just name the function according to the meaning of the value computed.

For example, `parcURIPath_Get`(,) returns the pointer to the n-th `PARC_URISegment` in a `PARCURI`. The `parcURIPath_Get`(f)unction does not increment references to the n-th `PARC_URISegment`.

If the caller must ensure that the segment is not deallocated later, it must acquire its own reference (via `parcURISegment_Acquire`()) and release that reference when it is no longer needed.

2.5 Acquire

This function increases the number of references to a PARC reference counted instance.

Type nameSpace*`_Acquire(`
    `const` *Type** `instance` A pointer to the PARC reference counted instance to acquire.

`)`

Returns:
*Type** A pointer to the original instance.

In this function, a given instance's reference count is incremented, but a new instance is not created. The acquired reference must be discarded by invoking `Release`(.)

```
PARCBuffer *buffer = parcBuffer_Allocate(10);

PARCBuffer *reference = parcBuffer_Acquire(buffer);

parcBuffer_Release(&buffer);
parcBuffer_Release(&reference);
```

Listing 1. Acquire Example

2.6 AssertValid

This function enforces instance validity when executed.

`void` *nameSpace*`_AssertValid(`
    `const` *Type ** `instance` A pointer to the instance to validate.
`)`

If the instance is a valid instance, return. Otherwise, display a meaningful message and catastrophically terminate the running program.

Valid means the internal state of the type is consistent with its required current or future behavior. This may include the validation of internal instances of encapsulated types.

2.7 BuildString

This function appends a string representation of the specified instance to the given `PARCElasticString`.

```
PARCElasticString * nameSpace_BuildString(
                Type *  instance    A pointer to the instance.
    PARCElasticString *  string     A pointer to a PARCElasticString instance.
)
```

Returns:

`NULL`	Cannot allocate memory.
non-`NULL`	The same `PARCElasticString` value as `instance`.

```
{
    PARCElasticString *result = parcElasticString_Create();

    parcBuffer_BuildString(instance, result);

    char *string = parcElasticString_ToString(result);
    printf("Hello: %s\n", string);
    parcMemory_Deallocate(&string);

    parcElasticString_Release(&result);
}
```

Listing 2. BuildString Example

2.8 Compare

This function compares instance *a* with instance *b* for order as defined by that type.

```
int nameSpace_Compare(
    const Type *  a   A pointer to instance a.
    const Type *  b   A pointer to instance b.
)
```

Returns:

< 0	The instance *a* is comparatively less than instance *b*.
$= 0$	The instances are comparatively equal.
> 0	The instance *a* is comparatively greater than instance *b*.

This function returns a negative integer, zero, or a positive integer if instance *a* is less than, equal to, or greater than instance *b*, respectively.

The implementor must ensure

- For all *a* and *b*: `Compare(a, b)) == -(Compare(b, a),`

- `Compare(a, b) > 0 && Compare(b, c) > 0 == Compare(a, c) > 0`,
- and for all *c*: `Compare(a, b) == 0` implies that `Compare(a, c) == Compare(b, c)`.

It is strongly recommended, but not strictly required that `(Compare(a, b) == 0) == Equals(a, b)`.

Generally, any type that implements the `Compare`(f)unction that violates this condition must clearly indicate this fact by stating, "This type has a natural ordering that is inconsistent with the Equals function."

```
{
    PARCBuffer *a = parcBuffer_Allocate(20);
    PARCBuffer *b = parcBuffer_Allocate(20);
    int result = parcBuffer_Compare(a, b);
}
```

Listing 3. Compare Example

2.9 Copy

The function creates an independent, deep-copy of the given instance.

Type * *nameSpace*_Copy(
    const *Type* * instance A non-NULL pointer to an instance to copy.
)

Returns:

NULL	Cannot allocate memory.
non-NULL	A pointer to a new instance.

A new instance is created as a complete, independent copy of the original such that `Equals(original, copy) == true`.

```
{
    PARCBuffer *buffer = parcBuffer_Allocate(20);
    PARCBuffer *copy = parcBuffer_Copy(buffer);

    parcBuffer_Release(&buffer);
    parcBuffer_Release(&copy);
}
```

Listing 4. Copy Example

2.10 Display

This function prints a human readable representation of the given instance.

```
void nameSpace_Display(
    const Type *  instance      A pointer to the instance to display.
          int     indentation   The indentation level to print.
)
```

Pretty-print the contents of the instance pointed to by `instance` in a way suited for human-readability.

The value of indentation is the ”nesting” level of the output. When an instance is a composite of other instances, the implementation may call each of the subcomponents Display function incrementing the indentation value by 1.

Implementations of Display should use the functions in `parc_DisplayIndented` to print the result.

```
{
    PARCBuffer *buffer = parcBuffer_Allocate(20);

    parcBuffer_Display(0, buffer);

    parcBuffer_Release(&instance);
}
```

Listing 5. Display Example

2.11 Equals

This function determines whether or not two instances are equal.

```
bool nameSpace_Equals(
    const Type *  instance   A pointer to instance X.
    const Type *  instance   A pointer to instance Y.
)
```

Returns:

`true`	The instances are equal.
`false`	The instances are not equal.

Two instances are equal if, and only if, the following equivalence relations on non-null instances are maintained:

- Reflexivity: for any non-null reference value x, `Equals(`x`, `x`)` must return true.

- Symmetry: for any non-null reference values x and y, `Equals(`x`, `y`)` must return true if and only if `Equals(`y`, `x`)` returns true.
- Transitivity: for any non-null reference values x, y, and z, if `Equals(`x`, `y`)` returns true and `Equals(`y`, `z`)` returns true, then `Equals(`x`, `z`)` must return true.
- Consistency: for any non-null reference values x and y, multiple invocations of `Equals(`x`, `y`)` consistently return true or consistently return false.
- For any non-null reference value x, `Equals(`x`, NULL)` must return false.

```
{
    PARCBuffer xa = parcBuffer_Allocate(20);
    PARCBuffer *y = parcBuffer_Allocate(20);

   if (parcBuffer_Equals(x, y)) {
        // true
    } else {
        // false
    }
}
```

Listing 6. Equals Example

2.12 HashCode

This function returns a value of type `PARCHashCode` value for the given instance.

```
PARCHashCode nameSpace_HashCode(
    Type *  instance   A pointer to the instance.
)
```

Returns:

`PARCHashCode` The hash code for the given instance.

The general contract of `HashCode`(i)s:

- Whenever it is invoked on the same instance more than once during an execution of an application, the `HashCode`(f)unction must consistently return the same value, provided no information used in a corresponding `Equals`(c)omparisons on the instance is modified.

  This value need not remain consistent from one execution of an application to another execution of the same application. If two instances are equal according to the `Equals`(f)unction, then calling the `HashCode`(f)unction on each of the two instances must produce the same integer result.

- It is not required that if two instances are unequal according to the `Equals`(f)unction, then calling the `HashCode`(m)ethod on each of the two objects must produce distinct integer results.

```
{
    PARCBuffer *buffer = parcBuffer_Allocate(10);
    uint32_t hashValue = parcBuffer_HashCode(buffer);

    parcBuffer_Release(&buffer);
}
```

Listing 7. HashCode Example

2.13 Init

This function initializes an instance of the supported type.

```
bool nameSpace_Init...(
    const Type *  instance   A pointer to the instance to initialize.
)
```

Sometimes it's advantageous to create a set or pool of reusable instances of a type, rather than going through the process of creating and releasing instances dynamically. In this case, the type's implementation may provide an initialization function which, given a pointer to an existing instance, initializes the instance to its freshly created state.

Implementations must take care to ensure that reinitializing an instance does not loose pointers to live objects held by the original instance.

If an instance multiple `Create` forms, it should have matching `Init` forms.

2.14 IsValid

This function determines whether or not an instance is valid.

```
bool nameSpace_IsValid(
    const Type *  instance   A pointer to the instance to validate.
)
```

If the instance is a valid instance, return true. Otherwise, return false.

Like `AssertValid`, valid means the internal state of the type is consistent with its required current or future behavior. This may include the validation of internal instances of encapsulated types.

This function is used to test whether or not an instance is valid before providing it to a function that would fail catastrophically if the instance were not valid.

2.15 Release

This function releases a previously acquired reference to the specified instance, and decrement the reference count for the instance.

```
void nameSpace_Release(
    Type **  pointer   A pointer to a pointer to the PARC Object instance to release.
)
```

The pointer to the instance is set to `NULL` as a side-effect of this function.

If the invocation causes the last reference to the instance to be released, the instance is deallocated and the instance's implementation will perform additional cleanup and release other privately held references.

The contents of the deallocated memory used for the PARC object are undefined. Do not reference the object after the last release.

```
{
    PARCBuffer *buffer = parcBuffer_Allocate(10);

    parcBuffer_Release(&buffer);
}
```

Listing 8. Release Example

2.16 ToString

This function produces a nul-terminated, C string representation of the specified instance.

```
char * nameSpace_ToString(
    Type *  instance   A pointer to the instance.
)
```

Returns:

`NULL`	Memory could not be allocated.
non-`NULL`	A null-terminated string that must be deallocated via `parcMemory_Deallocate`(.)

The non-null result must be freed by the caller via `parcMemory_Deallocate`(.)

```
{
    PARCBuffer *buffer = parcBuffer_Allocate(20);
    parcBuffer_PutArray(buffer, 12, "Hello World");
    parcBuffer_Flip(buffer);

    char *string = parcBuffer_ToString(buffer);

    if (string != NULL) {
```

```
        printf("%s\n", string);
        parcMemory_Deallocate(&string);
    } else {
        printf("Cannot allocate memory\n");
    }

    parcBuffer_Release(&instance);
}
```

Listing 9. ToString Example

2.17 ToJSON

This function produces a PARCJSON representation of the specified instance.

```
PARCJSON * nameSpace_ToJSON(
    Type *  instance    A pointer to the instance.
)
```

Returns:

NULL	Memory could not be allocated.
non-NULL	A valid PARCJSON instance that must be released via parcJSON_Release(.)

The non-null PARCJSON instance must be released by the caller via parcJSON_Release(.)

```
{
    PARCURI *uri = parcURI_Parse("http://parc.com");

    PARCJSON *json = parcURI_ToJSON(uri);

    // Display the JSON representation on stdout
    parcJSON_Display(json, 0);

    parcJSON_Release(&json);
    parcURI_Release(&uri);
}
```

Listing 10. ToJSON Example