The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

Copyright ©2002-2008 Derek M. Jones. All rights reserved.

6.5.2.2 Function calls

Constraints

function call

The expression that denotes the called function[78)] shall have type pointer to function returning **void** or returning an object type other than an array type. 997

Commentary

function designator converted to type

An occurrence of a function designator in an expression is automatically converted to pointer-to function type.

function declarator return type

identifier is primary expression if

A declaration of a function returning an array type is a constraint violation. Since identifiers denoting functions must be declared before they are referenced, the restriction on the object type being other than an array type is superfluous. Declaring a function to return a structure or union type, which contains a member having an array type, is sometimes used to get around the constraint on returning arrays from functions.

C++

5.2.2p3 *This type shall be a complete object type, a reference type or the type **void**.*

Source developed using a C++ translator may contain functions returning an array type.

Other Languages

Many languages usually distinguish between a *function* that has a return type and a *procedure* (or *subroutine*), which has no return type (they do not have an explicit **void** type). Some languages restrict functions to returning scalar types, while others allow functions to return any types, including function and array types.

function call arguments agree with parameters

If the expression that denotes the called function has a type that includes a prototype, the number of arguments shall agree with the number of parameters. 998

Commentary

That is, the prototype is visible at the point of call. It is not enough for it simply to exist in some translated file.

C++

C++ requires that all function definitions include a prototype.

5.2.2p6 *A function can be declared to accept fewer arguments (by declaring default arguments (8.3.6)) or more arguments (by using the ellipsis, **...** 8.3.5) than the number of parameters in the function definition (8.4). [Note: this implies that, except where the ellipsis (**...**) is used, a parameter is available for each argument.]*

A called function in C++, whose definition uses the syntax specified in standard C, has the same restrictions placed on it by the C++ Standard as those in C.

Other Languages

Those languages that support some form of function declaration that includes parameter information, often require that the number of arguments in the call agree with the declaration. Some languages (e.g., C++ and Ada) support default values for parameters. In these cases it is possible to omit arguments from the function call.

Coding Guidelines

function declaration use prototype ??

If the guideline recommendation specifying use of function prototypes is followed, all called functions will have a prototype in scope.

argument type may be assigned

Each argument shall have a type such that its value may be assigned to an object with the unqualified version of the type of its corresponding parameter. 999

Commentary

The assignment of the argument value to the parameter is similar to an initializer for an object definition. The qualifier is ignored. It only has meaning inside the body of the function that defines the parameter list. Developers sometimes use the term *assignment compatible*, however, the standard does not define this term.

initializer
initial value

C++

The C++ language permits multiple definitions of functions having the same name. The process of selecting which function to call requires that a list of viable functions be created. Being a viable function requires:

> *. . . , there shall exist for each argument an implicit conversion sequence that converts that argument to the corresponding parameter . . .*

13.3.2p3

A C source file containing a call that did not meet this criteria would cause a C++ implementation to issue a diagnostic (probably complaining about there not being a visible function declaration that matched the type of the call).

Other Languages

Other languages that contain some kind of type qualifier usually have a rule similar to C.

Semantics

1000 A postfix expression followed by parentheses `()` containing a possibly empty, comma-separated list of expressions is a function call.

operator
()

Commentary

Many developers do not think of `()` as an operator. In theory, a single token would be sufficient to indicate a function call (and the same for the `[]` operator). However, existing practice in other languages, and the bracketing effect of using two tokens (it removes the need to use parentheses in those cases where there is more than one parameter), were more important considerations.

An identifier having pointer-to function type that is not followed by parentheses returns the value of the pointer. No function is called.

The first step in improving the execution time of a program is often to measure how much time is spent in each function. Such measurements may or may not include the time spent in any functions that are themselves called by each function (some tools try to provide an estimate of the amount of time spent in any nested calls). Such measurements provide no context information; for instance, the time spent in function `D` may differ between the call chains `A`⇒`B`⇒`D` and `X`⇒`Y`⇒`D`. Call path refinement[14] provides measurements for each set of call paths. The amount of information gathered is much greater and more detailed, but it does allow special cases to be detected (which could then be appropriately specialized), such as the different call paths to `D`.

call graph

Understanding the relationship between functions in terms of which one calls, or is called, by another one provides a useful means of comprehending the structure of a program. A call graph is a practical way of displaying this calling/called information. The term *call graph* is usually taken to mean the static call graph. Information on the calls is obtained without executing the program. (Tools that build dynamic call graphs also exist.)

The C language contains two features that complicate the building of a call graph— a preprocessor and the ability to assign functions and later call them via the object assigned to.

The preprocessor is sometimes used to hide implementation details, such as calls to system libraries, that developers are not expected to be aware of, treating the macro invocation as if it were a function call. At other times a macro is used for efficiency reasons, and any function calls in its body are likely to be of interest to the developer. The preprocessor can also be used to provide conditional compilation. A function call may appear in the program image if certain macros are defined during translation. This raises the question of whether a call graph should only include function calls that are visible in the unpreprocessed source code, or should it only include function calls that appear in the token stream after preprocessing, or some combination of the two?

When a function is called through an object, having a pointer-to function type, the number of functions that could possibly be called is likely to be greater than one. Depending on the sophistication of the call graph analysis tool, it might show all functions having the pointed-to type of the object, all functions assigned to the object, or those functions that could possibly be pointed to at a given call site (although a recent study[24] was able to obtain some precise results on some of the GNU tools, using relatively inexpensive analysis).

An empirical study of static call graph tools by Murphy, Notkin, Griswold, and Lan[26,27] found that the call graphs they built were all different (in the set of called functions they each created). All tools individually listed called-functions that were not listed as called by any of the other tools, no call graph being a proper subset of another. One tool looked at the source code prior to preprocessing, one tool extracted information from program images that had been built with debugging information switched on, and the other tools operating in various in other ways.

C90

The C90 Standard included the requirement:

> *If the expression that precedes the parenthesized argument list in a function call consists solely of an identifier, and if no declaration is visible for this identifier, the identifier is implicitly declared exactly as if, in the innermost block containing the function call, the declaration*
>
> *extern int identifier();*
>
> *appeared.*

A C99 implementation will not perform implicit function declarations.

Other Languages

The use of `()` as the symbol, or operator, indicating a function call is widely used in computer languages. In some languages function calls are implicitly indicated by the type of an identifier and the context in which it occurs; use of `()` are not necessary. Other languages (e.g., Fortran) require the keyword **`call`** to appear before subroutine calls, but not before function calls that appear in an expression. In Lisp the name of the function being called appears as the first expression inside the parentheses. In functional languages it is even possible to invoke a function with an incomplete list of arguments, resulting in the partial evaluation of the called function. A few languages (e.g., ML) allow the function name to appear as an infix operator.

There is a standard for procedure calling: ISO/IEC 13886:1996 *Information technology— Language Independent Procedure Calling (LIPC)*. Quoting from the Introduction: “The purpose of this International Standard is to provide a common model for language standards for the concept of procedure calling.”.

Common Implementations

Implementation limits

The Implementation limits clause does not specify any minimum requirements on the depth of nested function calls. On most implementations the maximum function-call depth is set by the amount of storage available for the stack to grow into. Some processors have dedicated call stacks and the maximum nesting depth can be as low as 20.[29]

As processors have continued to evolve, the techniques for calling functions have come full circle. They started out simple, got very complex, and RISC brought them back to being simple (now they are starting to get complicated again). The complex processor instructions of the late 1970s and 1980s tried to perform all of the housekeeping actions associated with a function call. In practice, compiler writers often found it difficult to map the language semantics onto these complex instructions.[1000.1] In many cases it was possible to use alternative instructions that executed more quickly. (Some profiling studies showed that the complex instructions were rarely, if ever, encountered during program execution[33,38]). The RISC approach simplified

[1000.1] Measurements of assembly language usage by developers[8] has found that they use a subset of the instructions available to them.

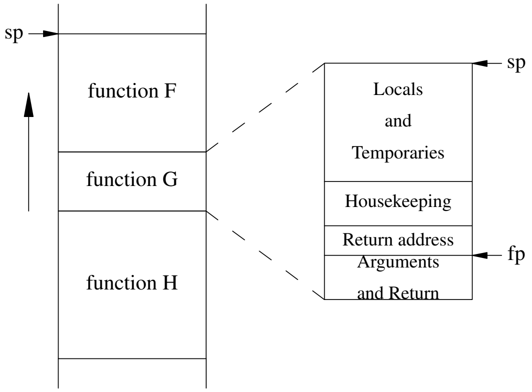

Figure 1000.1: Common storage organization of a function call stack.

everything, going back to very basic support for function calls (which is what low cost processors had been doing all the time, the resources needed to implement complex instructions being too costly).

The minimum requirement for a function-call machine code instruction is that it alters the flow of control (so that the processor starts executing the instructions of the called function), and that there is a mechanism for returning to the instructions after the call (so that execution can resume where it left off).

Processors use a variety of techniques for saving the return address. Some push it onto a stack or a specified storage location (the caller takes responsibility for saving the value); others load it into a register (the called function takes responsibility for saving it). The requirement to call functions recursively restricts the possible mechanisms that can be used to save the return address. An analysis by Miller and Rozas[25] showed that allocating storage on a stack to hold information associated with function calls was more time efficient than allocating it on the heap (even when the cost of garbage collection was minimal).

1026 function call recursive

In most implementations the housekeeping needed to store the arguments is performed in the function performing the call, and the housekeeping needed to allocate storage for locals is handled by the called function. A very common implementation technique is to use a function-call stack, each function call taking storage from this stack to satisfy its requirements, when called, and freeing it up on return. Because function calls properly nest, all operations are performed on the current top of stack.

register + offset

The address of the start of the stack storage for a function is known as its *frame pointer*. It is often held in a register and a *register* + *offset* addressing mode is used to access parameters and local objects. The *register* + *offset* addressing mode is not usually supported by DSP processors,[18] whose only indirect addressing modes are sometimes only pre-/postincrement/decrement of an address register performed as part of a storage access. A number of techniques have been proposed to optimize the generated machine code for processors having these following addressing mode characteristics:

- Emulating the *register* + *offset* addressing mode by using a frame pointer and arithmetic operations to create the address of the object is one possibility[21] (using a peephole optimizer on the resulting code).
- Using a floating frame pointer, arranging the ordering of objects in storage to maximize the likelihood that adjacent accesses in the generated code will be adjacent-locations in storage (so that an increment/decrement will create an address ready for the next access).[11]
- Using algebraic transformations on expression trees to obtain the least cost of access sequence.[32]
- Genetic algorithms have been proposed[20] as a general solution to the problem. (They are capable of adapting to the different numbers of addressing registers and the different autoincrement ranges available on different processors.)

Another technique is for translators to make use of the characteristics of the applications written for these processors; for instance, recursion is almost unknown, and allocating fixed-storage locations for local objects is often feasible. By building a call graph at link-time, linkers can deduce dependencies between calls and overlay storage for objects, defined in functions, whose functions are not simultaneously active.

The functional programming style of implementing algorithms creates a particular kind of function-call optimization[1] known as *tail calls*. Here the last statement executed in a function is a call to another function (a call to the same function is known as *tail recursion*). Such a recursive call can be replaced by a branch instruction to the start of the function that contains it. This tail recursion optimization idea can be extended to handle calls to other functions, as long as they occur as the last statement executed in a function. A call to another function can be replaced by a branch instruction (it may also be necessary to adjust the stack allocation for the amount of local storage). This optimization saves the creation of a new stack frame by reusing the existing one and can be worthwhile in target environments that have limited storage capacity.

Maintaining a general-purpose stack is an overhead that is not needed for some applications. Vendors of processors designed for use in freestanding environments have come up with a number of alternatives. For instance, the Microchip PIC 18CXX2 processor[23] call instruction pushes the return address onto a dedicated, 31-entry stack. Once this stack is full, the setting of the STVREN (stack overflow reset enable) flag determines the behavior; the processor will either reset or the current top of stack is overwritten with the latest return address. The STKFUL bit will have been set after entry 31 is used, allowing software to detect the pending problem.

One of the methods used by software viruses to gain control of a program is to overwrite a function's stack frame; for instance, changing the return address to point at malicious code that has been copied into some area of storage. A number of techniques to prevent such attacks have been proposed.[9, 12, 39]

There is often a significant difference in performance overhead between function calls that involve system calls (i.e., calls into the host operating system, such as `fread`, and `fwrite`) and other calls (a factor of 20 has been reported[30]). Merging multiple system calls into a single call can produce performance improvements.[30]

Coding Guidelines

cognitive switch

An occurrence of the `()` operator is likely to cause the reader to make at least one cognitive task switch. In some cases (e.g., a call to one of the trigonometric functions) readers may be able to immediately associate a return value with the call and its argument. In more complex cases, readers may have to put some thought into how the arguments passed will affect the return result. They may also need to take into account the fact that the call causes the values of some objects to be modified. There is nothing that can be done at the point of call to reduce the cognitive effort needed for these tasks.

The conditional `if (f)` is sometimes written when `if (f())` had been intended. However, this usage is a fault and these coding guidelines are not intended to recommend against the use of constructs that are obviously faults.

guidelines not faults

sequential nesting ()

macro function-like

The nesting of function calls (i.e., the result of one function call being used as the argument to another function call) raises a comprehension issue. In (which uses function-like macros to reduce the volume of code):

```
struct TREE_NODE {
                  int data;
                  struct TREE_NODE *left,
                                   *right;
                 };

#define LEFT_NODE(branch) ((branch)->left)
#define RIGHT_NODE(branch) ((branch)->right)

extern struct TREE_NODE *root;

void f(void)
{
struct TREE_NODE *leaf;

leaf = LEFT_NODE(RIGHT_NODE(LEFT_NODE(root))).data;
}
```

the expression has to effectively be read from the inside out (given that arguments occur to the right of the function name, the direction of reading is right-to-left). Studies have found that people have difficulty interpreting sentences containing more than two levels of nesting, for instance, “The mouse the cat the dog chased bit died.” Speakers of natural language have the same limitations as their listeners, so don’t produce such sentences. However, when writing, people do not have to deal with the constraints of realtime communications and consequently tend to write more complex sentences. Writers of source code rarely consider the issue of comprehension by future readers.

statement
syntax

Although there are often many ways of phrasing the same natural language sentence (“the mouse died which was bitten by the cat which the dog chased”), the C language rarely offers such flexibility at the expression level. One option is to break an expression into two parts; for instance:

```
left_right = RIGHT_NODE(LEFT_NODE(root));
leaf = LEFT_NODE(left_right).data;
```

An important issue to consider when breaking up an expression into separate components is the application semantics associated with each component. The identifier `left_right` suggests an implementation detail, not a semantic association (the issue of identifier name semantics is discussed elsewhere).

Identifier
semantic usability

While nested function calls may have a high-comprehension cost, there is no evidence to suggest that in general a guideline recommendation limiting the depth of nesting (and therefore requiring additional code to be written elsewhere) will have a worthwhile benefit (by any reduction in the required cognitive effort in this one case being greater than any increase in effort that occurs elsewhere). Similar human comprehension cost issues affect other operators.

member
selection
sequential nesting
*

Example

```
typedef int (*p_f)(int);

extern p_f f(float);

void g(void)
{
int loc = f(1.0)(2);
}
```

Usage

How frequent are function calls? The machine code instructions used to call a function may be generated by translators for reasons other than a function call in the source code. Some operators may be implemented via a call to an internal system library routine; for instance, floating-point operations on processors that do not support such operators in hardware. Such usage will vary between processors (see Figure **??**).

Table 1000.1: Static count of number of calls: to functions defined within the same source file as the call, not defined in the file containing the call, and made via pointers-to functions. Parenthesized numbers are the corresponding dynamic count. Adapted from Chang, Mahlke, Chen, and Hwu.[5]

Name	Within File	Not in File	Via Pointer
cccp	191 (1,414)	4 (3)	1 (140)
compress	27 (4,283)	0 (0)	0 (0)
eqn	81 (6,959)	144 (33,534)	0 (0)
espresso	167 (55,696)	982 (925,710)	11 (60,965)
lex	110 (63,240)	234 (4,675)	0 (0)
tbl	91 (9,616)	364 (37,809)	0 (0)
xlist	331 (10,308,201)	834 (8,453,735)	4 (479,473)
yacc	118 (34,146)	81 (3,323)	0 (0)

Table 1000.2: Percentage of function invocations during execution of various programs in SPECint92. The column headed *Leaf* lists percentage of calls to leaf functions, *NonLeaf* calls to nonleaf functions (the issues surrounding this distinction are discussed elsewhere). The column headed *Direct* lists percentages of calls where a function name appeared in the expression, *Indirect* is where the function address was obtained via expression evaluation. Adapted from Calder, Grunwald, and Zorn.[4]

Program	Leaf	Non-Leaf	Indirect	Direct	Program	Leaf	NonLeaf	Indirect	Direct
burg	72.3	27.7	0.1	99.9	eqntott	85.3	14.7	68.7	31.3
ditroff	14.7	85.3	1.0	99.0	espresso	75.0	25.0	4.0	96.0
tex	20.0	80.0	0.0	100.0	gcc	28.9	71.1	5.4	94.6
xfig	35.5	64.5	6.2	93.8	li	13.4	86.6	2.9	97.1
xtex	50.6	49.4	3.0	97.0	sc	29.1	70.9	0.1	99.9
compress	0.1	99.9	0.0	100.0	Mean	38.6	61.4	8.3	91.7

Table 1000.3: Count of instructions executed and function calls made during execution of various SPECint92 programs on an Alpha AXP21064 processor. *Function calls invoked* includes indirect function calls; *Instructions/Call* is the number of instructions executed per call; *Total I-calls* is the number of indirect function calls made; and *Instructions/I-call* is the number of instructions executed per indirect call. Adapted from Calder, Grunwald, and Zorn.[4]

Program Name	Instructions Executed	Function Calls Invoked	Instructions/Call	Total I-calls	Instructions/I-call
burg	390,772,349	6,342,378	61.6	8,753	44,644.4
ditroff	38,893,571	663,454	58.6	6,920	5,620.5
tex	147,811,789	853,193	173.2	0	–
xfig	33,203,506	536,004	61.9	33,312	996.7
xtex	23,797,633	207,047	114.9	6,227	3,821.7
compress	92,629,716	251,423	368.4	0	–
eqntott	1,810,540,472	4,680,514	386.8	3,215,048	563.1
espresso	513,008,232	2,094,635	244.9	84,751	6,053.1
gcc	143,737,904	1,490,292	96.4	80,809	1,778.7
li	1,354,926,022	31,857,867	42.5	919,965	1,472.8
sc	917,754,841	12,903,351	71.1	13,785	66,576.3
dhrystone	608,057,060	18,000,726	33.8	0	–
Program mean	497,006,912	5,625,468	152.8	397,233	14,614.1

Table 1000.4: Mean and standard deviation of call stack depth during execution of various programs in SPECint92. Adapted from Calder, Grunwald, and Zorn.[4]

Program	Mean	Std. Dev.	Program	Mean	Std. Dev.
burg	10.5	30.84	eqntott	6.5	1.39
ditroff	7.1	2.45	espresso	11.5	4.67
tex	7.9	2.71	gcc	9.9	2.44
xfig	11.6	4.47	li	42.0	14.50
xtex	14.2	4.27	sc	6.8	1.41
compress	4.0	0.07	Mean	12.0	6.29

1001 The postfix expression denotes the called function.

Commentary

The symmetry seen in the `[]` operator does not also apply in the case of the `()` operator. The left operand always denotes the called function.

array subscript
identical to

An analysis by Zhang and Ryder[40] found that, for programs using only a single level of pointer-to function indirection (the simplest kind of pointers), statically determining the possible called functions at a call site is NP-hard.

Other Languages

Many languages do not support pointers to functions. For these languages the postfix expression is required to be an identifier.

Coding Guidelines

When the postfix expression is not an identifier denoting the name of a function but an object having a pointer-to function type it can be difficult to deduce which function is being is called. The possibility that the value has been cast from a different pointer-to function type complicates matters even more. Some coding guideline documents adopt a simple solution to the problem of knowing which function is actually being called. They ban the use of nonconstant function pointers.

If the use of nonconstant function pointers is prohibited, what are the alternative constructs available to developers? A sequence of *selection-statement*s could be used to select the function that is to be called. These alternatives do simplify the task of deducing the set of functions that could be called. However, they also potentially increase the effort needed to maintain the source. If any of the functions called changes, it could be necessary to edit more than one location in the source code. (All the **if** statements or **switch** statements referencing the changed function will need to be looked at, unless it is possible to locate the sequence of **if** statements, or **switch** statements in a macro, or inline function.)

inline function

Declaring an array of pointers, initialized with function pointers, creates a single maintenance point in the source.

While use of nonconstant pointers to functions may increase the cognitive effort needed to comprehend source code, the use of alternative constructs could not be said to reduce the effort, and could increase it.

Example

```
extern int f_1(void);
extern int f_2(void);

extern int (* const p_f[])(void) = { f_1, f_2 };

void g(void)
{
p_f[0]();
}
```

Table 1001.1: Static count of functions defined, library functions called, direct and indirect calls to them and number of functions that had their addresses taken in SPECint95. Adapted from Cheng.[6]

Benchmark	Lines Code	Functions Defined	Library Functions	Direct Calls	Indirect Calls	& Function
008.espresso	14,838	361	24	2,674	15	12
023.eqntott	12,053	62	21	358	11	5
072.sc	8,639	179	53	1,459	2	20
085.cc1	90,857	1,452	44	8,332	67	588
124.m88ksim	19,092	252	36	1,496	3	57
126.gcc	205,583	2,019	45	19,731	132	229
130.li	7,597	357	27	1,267	4	190
132.ijpeg	29,290	477	18	1,016	641	188
134.perl	26,874	276	72	4,367	3	3
147.vortex	67,205	923	33	8,521	15	44

The list of expressions specifies the arguments to the function. 1002

Commentary

This defines what the arguments to a C function are.

Usage

function call number of arguments

Usage information on the number of arguments in calls to functions is given elsewhere.

An argument may be an expression of any object type. 1003

Commentary

argument 999 type may be assigned

If the visible function declaration does not include a prototype, this sentence makes passing an argument having an incomplete type undefined behavior. The case where a function prototype is in scope is covered by an earlier constraint.

Only object types have values and a value is needed to assign to the parameter.

Other Languages

Some languages support the use of labels as arguments. While others, particularly functional languages, support the use of types as arguments.

Common Implementations

base document

The undefined behavior of most translators, on encountering an argument that does not have an object type in a call to a function whose declaration does not include a prototype, is to issue a diagnostic. The base document did not support the passing of structure or union types as parameters; a pointer to them had to be passed.

Usage

Information on parameter types is given elsewhere (see Table **??**).

Table 1003.1: Occurrence of various argument types in calls to functions (as a percentage of argument types in all calls). Based on the translated form of this book's benchmark programs.

Type	%	Type	%
struct *	26.8	**void** *	4.0
int	16.5	**union** *	3.4
const char *	9.7	**unsigned char**	2.5
char *	8.4	**enum**	2.1
other-types	8.0	**unsigned short**	1.9
unsigned int	7.1	**const void** *	1.8
unsigned long	6.3	**long**	1.4

In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument.[79)] 1004 function call preparing for

Commentary

This form of argument-passing is known as *call by value*. The standard does not specify any order for the evaluation of the arguments. 1025 function call sequence point

In some cases there may not be a parameter to assign the argument to. C supports the passing of variable numbers of arguments in a function call. The C90 Standard introduced the variable arguments mechanism for accessing the values of these arguments. Prior to the publication of C90, a variety of implementation-specific techniques were used by developers. These techniques relied on knowing the argument-passing conventions (which invariably involved taking the address of a parameter and incrementing, or decrementing, walking through the stack to access any additional arguments).

C++

The C++ Standard treats parameters as declarations that are initialized with the argument values:

> *When a function is called, each parameter (8.3.5) shall be initialized (8.5, 12.8, 12.1) with its corresponding argument.* 5.2.2p4

The behavior is different for arguments having a class type:

> *A class object can be copied in two ways, by initialization (12.1, 8.5), including for function argument passing (5.2.2) and for function value return (6.6.3), and by assignment (5.17).* 12.8p1

This difference has no effect if a program is written using only the constructs available in C (structure and union types are defined by C++ to be class types).

Other Languages

Many languages support a form of argument-passing known as *call by reference*. Here the address of the argument is passed and become the address through which parameter accesses occur. Inside the called function the parameters have the same type as the arguments; they are not pointers to that type. An assignment to the parameter stores the value into the corresponding object passed as the argument. The parameter is effectively an alias for the argument.

A similar, but slightly different, argument-passing mechanism is in/out passing. Here the argument value is copied into the parameter, but modifications to the parameter do not immediately affect the value of the object used as the argument. Just before the function (or procedure) returns, the current value of the parameter is copied into the corresponding object argument. In this case the parameters are not aliases of the corresponding argument objects, but values can still be passed back to the calling function.

Java passes arguments that have primitive type (the C arithmetic type) by value. Arguments having any other type are passed by reference.

Some languages do not (e.g., Haskell, Miranda, sometimes Lisp, Scheme) evaluate the arguments to a function before it is called. They are only evaluated on an as-needed basis, inside the called function, when they are accessed. Such a mechanism is known as *lazy*, or *normal-order evaluation*.

Algol 60 used what is known as *call by name* argument-passing. When this language was designed, developers were familiar with the use of assembly language macros. The arguments to these macros were expanded out in the body of the macro, just like the C preprocessor. The designers of Algol 60 decided that arguments to functions should have the same semantics. Algol 68, and other languages derived from Algol 60, did not duplicate this design decision.

Some languages (e.g., Ada, C++, and Fortran 90) support *named* parameters. Here the name of the parameter appears in the argument list to the left of the argument value. By using names it is possible to list the arguments in any order rather than relying on a default order implicit in the function definition.

Ichbiah, Barnes, Firth, and Woodger[17] discuss the rationale behind the selection of argument-passing mechanisms in Ada.

Common Implementations

One of the simplest, and commonly seen, techniques is to push arguments onto the stack used to hold the function return address and local object definitions. The addresses of each parameter in the called function are the stack locations holding the corresponding argument.

Different methods of passing arguments are seen on different processors and in different operating systems (Johnson[19] discusses the original thinking behind the C calling sequence). In a hosted environment there are usually strong commercial incentives for all translator vendors to use the same conventions (it enables calls to libraries written using different translators and languages to be intermixed). Since the early 1990s it has become common practice for processor vendors to publish a document called an ABI (Applications Binary Interface).[16, 34–36, 41] One of the details specified by this document are the argument-passing conventions.

Calling conventions can also depend on how functions are defined. Functions defined using prototypes and the static storage class are good candidates for aggressive optimization of their calling sequence. Because they are not externally visible, the translator does not have to worry about calls where the prototype might not be visible. In the case of functions that are externally visible, translators have to play safe and follow documented calling conventions (unless using a link-time optimizer[28]).

The IAR PICmicro compiler[15] supports the **__bankN_func** specifier (where N is a decimal literal denoting particular storage banks), which can be used by developers to indicate, to the translator, which storage bank should be used to pass arguments in calls to functions (defined using this specifier).

function call
number of arguments

Analysis of function calls shows that it is rare for many arguments to be passed. This usage suggests the optimization of passing the first few arguments in registers and pushing any subsequent arguments onto the stack. In practice this is an optimization, which at first sight looks very simple, but turns out to be potentially very complex and sometimes not even an optimization at all. For instance, if the called function itself calls a function, the contents of the reserved registers need to be saved before the new arguments for the call are loaded into them.

register
function call
housekeeping

There is one task that needs to be performed, before performing a function call, that the standard does not mention. The contents of registers holding values that will be used after the call returns need to be saved. Implementations tend to adopt one of two strategies for saving these registers— caller saves registers (it only needs to save the ones that contain values that are accessed after the call returns) or callee saves registers (it only needs to save the registers it knows it modifies). Caller saves is commonly seen. An analysis by Davidson and Whalley[10] compared various methods and concluded that a hybrid scheme was often the best. It has been proposed that processors include an instruction that, when executed, tells the processor that the contents of a particular register will not be read again before another value is loaded, so called *dead value* information.[22] An overall performance increase of 5% is claimed from not having to save/restore values across function calls or process context switches.

In the 1990s researchers started to investigate link-time optimizations. The linker has information on register usage available to it, by function, across all translation units; thus, it is possible to work out exactly, not make worst-case assumptions, which registers need to be saved and restored. Measurements from one such tool[28] showed context-sensitive interprocedural register liveness analysis reducing the total number of load instructions performed during program execution by 2.5% to 5% (the context-insensitive figure was 2.5–3%).

Processor vendors have also attempted to design instructions that reduce the overhead associated with saving register contents. Some CISC processors[7] include call instructions which automatically save the contents of some registers. An alternative processor architectural technique, adopted by Sun in their SPARC architecture[37] and more recently by Intel in the Itanium processor family,[31] is to use overlapping register windows. Here the processor supports a large internal register file (often 128 or more registers) of which a subset (usually 32, the *register window*) are available to the program. Execution of a *call* instruction causes the register window to slide up the internal register file such that, for instance, 16 *new* registers become

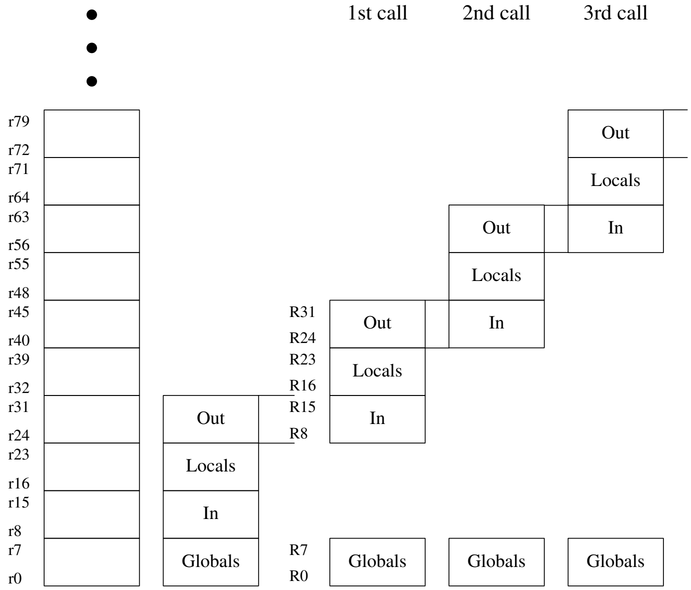

Figure 1004.1: A processor's register file (on the left) and a mapping to register windows for registers accessible to a program, after 0, 1, 2, and 3 *call* instructions have been executed. The mapping of the first eight registers is not affected by the *call* instruction.

available and 16 registers contents are *saved* (instructions always refer to registers numbered between 0 and 31, in the case of registers numbered between 8 and 31 the actual internal register referenced depends on the current depth of call instructions). Up to eight arguments in the *output* registers become available in the called function via its parameters in the *input* registers (see Figure 1004.1). Executing a return instruction slides the register window back, making the previous register contents available again. If there are sufficient nested call instructions the register file fills up and some of its contents have to be spilled to storage (a time consuming process). The performance advantage of registers windows comes from typical program behavior, where call depth does not vary a great deal during program execution (i.e., register file contents rarely have to be spilled and reloaded from storage). The optimal size of register window to use involves trade-offs among several hardware design factors.[13]

1005 If the expression that denotes the called function has type pointer to function returning an object type, the function call expression has the same type as that object type, and has the value determined as specified in 6.8.6.4.

Commentary

This defines the type of the function call expression (to which the `()` operator is applied) and the type of its value (created by the evaluation of an expression appearing in a **`return`** statement in the called function's body).

C90

A rather embarrassing omission from the original C90 Standard was the specification of the type of a function call. This oversight was rectified by the response to DR #017q37.

C++

Because C++ allows multiple function definitions having the same name, the wording of the return type is based on the type of function chosen to be called, not on the type of the expression that calls it.

5.2.2p3

The type of the function call expression is the return type of the statically chosen function . . .

Other Languages

Some languages use the convention that a function call always returns a value, while procedure (or subroutine) calls never return a value.

Common Implementations

Although not explicitly specified in the C90 Standard, all implementations effectively used the above definition (and did not regard the omission of a definition in the standard as implying undefined behavior).

Table 1005.1: Occurrence of various return types in calls to functions (as a percentage of the return types of all function calls). Based on the translated form of this book's benchmark programs.

Type	%	Type	%
void	35.8	**union** *	3.2
int	30.5	**unsigned long**	3.1
struct *	9.1	**char** *	3.1
void *	6.3	**unsigned int**	2.1
other-types	5.2	**char**	1.6

Otherwise, the function call has type **void**. 1006

Commentary

In this case it has no value.

C++

C++ also supports functions returning reference types. This functionality is not available in C.

function result attempt to modify

If an attempt is made to modify the result of a function call or to access it after the next sequence point, the behavior is undefined. 1007

Commentary

The result referred to here is any actual storage returned by the function call. This situation can only occur if the function return type is a structure or union type (see Examples that follow).

C90

This sentence did not appear in the C90 Standard and had to be added to the C99 Standard because of a change in the definition of the term lvalue.

lvalue

C++

The C++ definition of lvalue is the same as that given in the C90 Standard, however, it also includes the wording:

5.2.2p10 *A function call is an lvalue if and only if the result type is a reference.*

In C++ it is possible to modify an object through a reference type returned as the result of a function call. Reference types are not available in C.

Common Implementations

The implementation of functions that return a structure or union type usually involves the use of, translator allocated, temporary storage. This storage is usually allocated in the stack frame of the calling function and passed to the called function as an additional, hidden from the developer, argument. The C Standard only requires that the lifetime of the temporary storage used to hold the return value exist until the next sequence point after the return. Reuse of any part of this temporary storage by developers (e.g., to hold other temporary values) could overwrite any previous contents.

Coding Guidelines

By continuing to access the result of a function call, developers are relying on undefined behavior for what is essentially an efficiency issue (they are manipulating the returned result directly rather than assigning it to a declared object). However, on the basis that occurrences of this usage are likely to be rare, a guideline recommendation is not considered cost effective.

Example

In the following the temporary used to hold the value returned by `f` may, or may not be used for other purposes after the function has returned.

```
struct S {
          int m1;
          unsigned char m2[10];
         };

extern struct S f(void);

void g(void)
{
int *pi = &(f().m1); /* Get hold of the address of part of the result. */
unsigned char uc_1,
              uc_2;

*pi = 11; /* Undefined behavior. */

/*
 * In the following there is a sequence point after the comma operator.
 */
uc_1 = f().m2[uc_2=0, 2]; /* Also undefined, and probably surprising to readers. */

/*
 * Depending on the unspecified order of evaluation, the
 * following may also be undefined.
 */
uc_1 = f().m2[f().m1];
}
```

called function
no prototype

1008 If the expression that denotes the called function has a type that does not include a prototype, the integer promotions are performed on each argument, and arguments that have type **`float`** are promoted to **`double`**.

Commentary

This sentence applies when an old-style function declaration is visible at the point of call. If no prototype is visible, there is no information on the expected type of the argument available to the translator. On many processors the minimum-size object that can be pushed onto the stack (the most common argument-passing mechanism) has type **`int`**. The integer promotion rules for arguments match how their value is likely to be passed to the function, even if they had a type whose size was smaller than **`int`**. This behavior also matches the case where an argument expression contains operators, causing the integer promotions to be performed on the operands.

int
natural size

A consequence of performing the integer promotions on arguments is that the translator needs to generate matching machine code for accessing the parameters in the called function. It is possible that it may need to implicitly convert a value back to its original type.

1011 function definition
ends with ellipsis

The conversion of real type **`float`** to **`double`** does not apply to complex types. The following wording was added to the Rationale by the response to DR #206:

Rationale

float _Complex is a new type with C99. It has no need to preserve promotions needed by pre-ANSI-C. It does not break existing code.

C++

In C++ all functions must be defined with a type that includes a prototype.

A C source file that contains calls to functions that are declared without prototypes will be ill-formed in C++.

Other Languages

Most languages do not contain multiple integer types. Those languages that do contain multiple floating-point types do not usually specify that arguments having these types need be converted prior to the call.

Coding Guidelines

function declaration use prototype ??

If the guideline recommendation specifying use of function prototype declarations is followed the integer promotions will not be performed.

---

default argument promotions

These are called the *default argument promotions*. 1009

Commentary

integer promotions

This defines the term *default argument promotions*. They are a superset of the integer promotions in that they also promote the type **float** to **double**.

C++

The C++ Standard always requires a function prototype to be in scope at the point of call. However, it also needs to define default argument promotions (Clause 5.2.2p7) for use with arguments corresponding to the ellipsis notation.

Coding Guidelines

The term *argument promotions*, or the *arguments are promoted*, are commonly used by developers. There is no obvious benefit in investing effort in changing this existing usage.

---

arguments same number as parameters

If the number of arguments does not equal the number of parameters, the behavior is undefined. 1010

Commentary

Prior to the C90 Standard no prototypes using the ellipsis notation were available. However, developers still expected to be able to pass variable numbers of arguments to functions. The C Committee added the ellipsis notation for passing variable numbers of arguments to functions. However, the Committee could not make it a constraint violation for the number of arguments passed to disagree with the number of parameters in a nonprototype function definition. It would have invalidated a large body of existing source code.

Other Languages

Most languages require that the number of arguments agree with the number of parameters. Some classes of languages (e.g., functional) are specifically intended to support function invocations where fewer arguments than parameters are passed.

Common Implementations

Passing the incorrect number of parameters is both a common mistake, a programming technique that continues to be used by some developers, and something that occurs within long-existent source code. Being able to handle this case is usually seen as commercially essential for an implementation.

If the argument-passing conventions are such that the leftmost argument is nearest the top of the stack (if the stack grows down it will have the lowest address, if the stack grows up it will have the highest address), then the address of a particular parameter in the called function does not have a dependency on the number of arguments given in the call. The address of the parameter is known at translation time and can be accessed by indexing off the stack pointer (or the frame pointer, if the implementation uses one, see Figure 1000.1).

There are a number of implementation techniques for passing arguments to functions. These techniques all rely, to some degree, on the caller performing some of the housekeeping. As well as assigning arguments to parameters, this housekeeping also usually includes popping them off the stack when the called function returns (assuming this is the argument-passing convention used).

Coding Guidelines

Given the guideline recommendation dealing with the use of function prototypes this situation is only likely to occur when developers are modifying existing code that does not use function prototypes. Mismatches between the number and types of arguments in function calls, when no prototype is visible, and the corresponding function definition are a very common source of programming error (experience shows that once developers who use the non-prototype form are introduced to the benefits of using prototypes, they rarely want to switch back to using the non-prototype form). These mismatches are not required to be diagnosed by a translator if no prototype is visible at the point of call.

?? function declaration use prototype

undefined behavior

Developers do not usually provide the incorrect number of arguments in a function call on purpose (an existing developer practice is to call the Unix system function open with 2 or 3 arguments depending on the value of the second). A guideline recommending that the expected number of arguments always be passed serves no purpose. The solution to the underlying problem is to use prototypes. The extent to which it is cost effective to modify existing code to use prototypes is discussed elsewhere.

prototypes cost/benefit

1011 If the function is defined with a type that includes a prototype, and either the prototype ends with an ellipsis (`, ...`) or the types of the arguments after promotion are not compatible with the types of the parameters, the behavior is undefined.

function definition ends with ellipsis

Commentary

This C sentence applies when calling a function, defined using a prototype, when an old-style declaration (not the prototype) is visible at the point of call.

When translating a function whose definition uses a prototype an implementation knows the types of the parameters and can make use of this information. In particular it need not implicitly convert a reference to a parameter that does not have a type that is the same as its promoted type. It may also choose to assign different storage addresses to such parameters, when declared with and without the use of prototype notation.

1019 function call prototype visible

There are advantages to requiring implementations to make sure this case is well defined. (It would enable developers to slowly migrate their source code over to using prototypes, changing definitions to use prototypes, and eventually getting around to ensuring that a prototype is visible from all calls to that function.) The disadvantage of such an approach is that it would have tied implementations to backward compatibility, removing the possibility of most of the optimizations that prototypes enable translators to perform.

The occurrence of an ellipsis in a function's definition does not automatically require an implementation to guarantee that all calls to that function have well-defined behavior. In some implementations the argument/parameter mechanism used when a function is defined using ellipsis is completely different from when an ellipsis is not used. If, at the point of call, it is not known that the function definition contains an ellipsis (because no prototype is visible), the generated code may be incorrect.

Other Languages

Most languages either specify that some form of prototype declarations always be used, or do not provide any mechanism for specifying the types of parameters. However, it is not uncommon for a language to evolve into supporting both kinds of function definition (provided it originally supported the nonprototype form only). For instance, the ability to declare subroutines taking arguments was added in release 5.003 of Perl.

Common Implementations

There are two commonly seen cases: (1) the parameter has an integer type whose rank is less than that of **`int`**; (2) the type of the argument, after promotion, is not compatible with the parameter type. It is not uncommon for implementations to give unexpected results when reading a parameter having a character type in a function defined using a prototype, when that function has been called in a context where no prototype is visible. In some cases the machine code generated by a translator for the called function, accesses the value of the parameter using an instruction that reads more than eight bits (the typical number of bits in a character type). The assumption being made by the translator is that at the point of call the argument will have been converted to the character type, bringing it into the representable range of that type (the typical implementation behavior for all character types). If no prototype is visible at the point of call, this conversion will not have occurred, and it is possible for a value outside of the representable range of a character type to be assigned to the parameter.

Assuming that arguments are passed by pushing values onto a stack, the number of bytes pushed onto the stack will depend on the number of bytes in the promoted argument type. If the types **int** and **long**, for instance, have the same size, then mixing their usage in arguments is likely to have no surprising results. However, if these two types have different sizes, it is likely that after an argument having the other type is evaluated, any subsequent arguments (which will depend on the order of evaluation) will be pushed at addresses that are different from where their corresponding parameters expect them to be.

Coding Guidelines

If a function is defined using a prototype, the only reason for not having this prototype visible at the point of call is that the translator being used does not support prototypes (i.e., it is different from the one that translated the function definition). The guideline recommendation dealing with the use of function prototypes is applicable here.

function declaration use prototype ??

Example

The handling of parameters inside a function definition can be different from when a prototype is used than when it is not used. In:

file_1.c

```
int f_np(a, b)
signed char a;
int b;
/*
 * Function body assumes that any call will perform the default argument
 * promotions.  So it will insert casts, where necessary, on parameters.
 */
{
return a + b;
}

int f_pr(signed char a, int b)
/*
 * Function body assumes that a prototype is visible at the point of call.
 * Hence the arguments will have been cast at the point of call.  So there
 * is no need to insert an implicit conversion on any parameter reference.
```

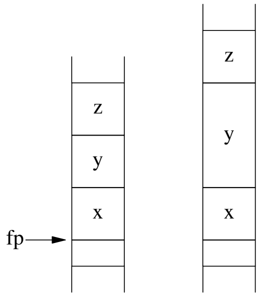

Figure 1011.1: An example of the impact, on relative stack addresses, of passing an argument having a type that occupies more storage than the declared parameter type. For instance, the offset of z, relative to the frame pointer *fp*, might be changed by passing an argument having a type different from the declared type of the parameter y (this can occur when there is no visible prototype at the point of call to cause the type of the argument to be converted).

```
 */
{
return a + b;
}
```

file_2.c

```
extern int f_np();
extern int f_pr();

extern signed char sc;
extern int i;
extern int glob;

int g(void)
{
glob = f_np(sc, i);
glob = f_pr(sc, i);
}
```

at the call to `f_np` there is no visible information on the function parameter types. If there had been, the first argument would have been cast to type **`signed char`**.

So what is the value of the parameter a inside the function `f_pr`? No implicit conversion need occur. If machine code to load one byte is generated by the translator, the value will only ever be in the range of the type **`signed char`**. However, if the generated code loads a larger unit of storage, it may be quicker to load a word, making the assumption that the top bits are all zero rather than to load a byte and zeroing the top bits. In the load word case the possible range of values is that of type **`int`**, not **`signed char`**.

1012 If the function is defined with a type that does not include a prototype, and the types of the arguments after promotion are not compatible with those of the parameters after promotion, the behavior is undefined, except for the following cases:

argument in call incompatible with function definition

Commentary

The rationale for this compatibility requirement is the same as that for the prototype case (the only difference is that the compatibility check is against the promoted type of the parameter). The call and definition need to agree on how the type of the argument affects how and where values are passed to the called function.

1011 function definition ends with ellipsis

C90

The C90 Standard did not include any exceptions.

C++

All functions must be defined with a type that includes a prototype.

A C source file that contains calls to functions that are declared without prototypes will be ill-formed in C++.

Common Implementations

The undefined behavior for these cases of all known C90 implementations is as defined in C99.

Coding Guidelines

Passing an argument having the incorrect type is a fault and these coding guidelines are not intended to recommend against the use of constructs that are obviously faults. The two cases listed in the following C sentences are intended to codify practices that are relatively common in existing code, not for use in newly written code. Adhering to the guideline recommendation on using function prototypes enables translators to implicitly perform any necessary conversion or to issue a diagnostic is none is available.

guidelines not faults

?? function declaration use prototype

Example

Here both the function definition and the point of call must agree on how they treat the type of the argument. In the following:

file_1.c

```
#include <stdio.h>

void f(a)
short a;
{
printf("a=%d\n", a);
}
```

file_2.c

```
extern void f();

void g(void)
{
f(10);  /* First call. */
f(10L); /* Second call. */
}
```

the argument passed in the first call to `f` is compatible with the promoted type of the parameter in the definition of `f`. In the second call the argument is not compatible, even though the value is representable in the promoted parameter type (and also the unpromoted parameter type).

footnote 78

78) Most often, this is the result of converting an identifier that is a function designator. 1013

Commentary

A self-evident, to experienced developers, piece of information is carried over from the C99 document.

C++

The C++ language provides a mechanism for operators to be overloaded by developer-defined functions, creating an additional mechanism through which functions may be called. Although this mechanism is commonly discussed in textbooks, your author suspects that in practice it does not account for many function calls in C++ source code.

Other Languages

This statement is true in most programming languages.

Common Implementations

The implicit conversion of a function designator to a pointer-to function is optimized away by most implementations, generating a machine code instruction to perform a function call to a known (at link-time) address.

Usage

In most programs an identifier is converted in more than 99% of cases, although a lower percentage is occasionally seen (see Table 1001.1).

footnote 79

79) A function may change the values of its parameters, but these changes cannot affect the values of the arguments. 1014

Commentary

operator () 1000

The storage used for parameters is specific to the function invocation. The final value of a parameter is never copied back to the corresponding argument. The storage used for parameters is released when the invocation of the function, that defines, them returns.

C++

The C++ reference type provides a call by address mechanism. A change to a parameter declared to have such a type will immediately modify the value of its corresponding argument.

This C behavior also holds in C++ for all of the types specified by the C Standard.

Other Languages

In languages that support various forms of argument passing (e.g., call by value, call by reference, or **OUT** style parameters) changes to the value of a parameter may or may not affect the value of the argument, or may be a violation of the language semantics (e.g., parameters defined using **IN** in Ada).

1004 function call preparing for

Coding Guidelines

Some coding guideline documents do not permit parameters to be modified within the function definition. They are considered to be read-only objects. The rationale for such a guideline often draws parallels with other languages, where a modification of a parameter also modifies the corresponding argument. Should a set of C coding guidelines take into account the behavior seen in other languages, and if so, which others languages need to be considered? Would a guideline recommendation against modifying parameters reduce or increase faults in programs? would it reduce or increase the effort needed to comprehend a function? Are the alternatives more costly than the original problem, if there is one? Without even being able to start answering these questions, there is no point in attempting to create a guideline recommendation.

1015 On the other hand, it is possible to pass a pointer to an object, and the function may change the value of the object pointed to.

Commentary

Parameters can have any object type (which include pointer types). Assigning the value of an argument, having a pointer type, to a parameter, having a pointer type, has the same effective semantics as assigning any pointer value to an object. Both pointers point at the same object. A modification of the value of the pointed-to object through either pointer reference will be visible via the other pointer reference.

C++

This possibility is not explicitly discussed in the C++ Standard, which supports an easier to use mechanism for modifying arguments, reference types.

Other Languages

This statement is generally true in languages that support pointers. In some languages (e.g., function languages) it is never possible to have two pointers (or references as they are usually known) to the same object. An assignment, or argument, would cause the pointed-to object to be copied. The pointer assigned would then point at a different object that had the same value.

Coding Guidelines

One of the issues overlooked by coding guideline documents which recommend against the use of pointers, is that argument-passing in C uses pass by value. If pointers cannot be used, the only way for a function to pass information back to its caller is via a returned value, or by modifying file scope objects. The alternatives could be worse than what they are replacing.

Usage

Pointer types are the most commonly occurring kind of parameter type (see Table **??**).

1016 A parameter declared to have array or function type is adjusted to have a pointer type as described in 6.9.1.

Commentary

This adjustment is made to the types of the parameters in a function definition. The arguments are similarly adjusted.

function definition syntax

array converted to pointer

function designator converted to type

Coding Guidelines

The benefit of defining parameters using array types rather than the equivalent pointer type comes from the additional information provided (the expected number of elements) to tools that analyze source code (enabling them to potentially be more effective in detecting likely coding faults). There is new functionality in C99 that allows developers to specify a lower bound on the number of elements in the object passed as an argument and to specify that after conversion, the pointer rather than the pointed-to object is qualified.

function declarator static

— one promoted type is a signed integer type, the other promoted type is the corresponding unsigned integer type, and the value is representable in both types; 1017

Commentary

Many argument expressions are integer constants, often having type **signed int**. A requirement that all such arguments be explicitly cast, or contain a suffix, in those cases where the corresponding parameter has type **unsigned int** was considered onerous. This exception allows unsuffixed integer constants to be used as arguments where the corresponding parameter has type **unsigned int**.

This exception requires that the rank of the signed and unsigned integer types be the same. It does not provide an exception for the case where an integer constant argument has a different type from the corresponding parameter type, but is representable in the parameter type. There are two requirements specified elsewhere that ensure implementations meet this requirement: (1) identical values of corresponding signed/unsigned integer types, and (2) storage and alignments requirement.

signed integer corresponding unsigned integer

footnote 31

Other Languages

Most languages do not contain both signed and unsigned integer types.

Coding Guidelines

integer constant usage

Most integer constants that appear in source code have small positive values (98% of decimal literals and 88% of hexadecimal literals are less than 32,768) of type **signed int** (only 2% of constants are suffixed). In these cases there is no obvious benefit to using a cast/suffix when the corresponding parameter has an unsigned type. For other kinds of arguments, more information on the cost/benefit of explicit casts/suffixes for arguments is needed before it is possible to estimate whether any guideline recommendation is worthwhile.

Example

In the following values of type **unsigned int** and **int** are passed as arguments. Prior to C99, the C90 Standard specified that this was undefined behavior.

```
extern unsigned int ui;

extern void f();

void g(void)
{
f(ui);
f(22);
}
```

— both types are pointers to qualified or unqualified versions of a character type or **void**. 1018

Commentary

footnote 39

Normative references

Requiring that arguments having type pointer-to unqualified character type be explicitly cast to the parameter type (to remove an undefined behavior that another part of the standard specifies, although not normatively) was regarded as being excessive. A commonly used argument of type pointer-to unqualified character type is a string literal. Some functions in the C library have parameters whose type is `const char *`. A large amount of existing code uses the pre-C Standard definition of these functions, which does not include the **const** qualifier.

Other Languages

Some languages have generic pointer types that can be passed as arguments where the corresponding parameter type is a pointer to some other type.

Coding Guidelines

What kind of pointer argument types and pointer parameter types, in an old-style function definition, are likely to occur in existing code? It is unlikely that parameters having a pointer type will be qualified (these were introduced into C at the same as prototypes), and they are probably more likely to be pointers to character type than pointer to **void**. The only constant pointer type is the null pointer constant, whose type is not under developer control. If this exception case is invoked, the argument type is likely to either have a pointer-to qualified type or be a pointer to **void**.

More information on the cost/benefit of explicit casts, for arguments, is needed before it is possible to evaluate whether any guideline recommendation is worthwhile.

1019 If the expression that denotes the called function has a type that does include a prototype, the arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters, taking the type of each parameter to be the unqualified version of its declared type.

function call prototype visible

Commentary

When a prototype is visible, argument conversion occurs at the point of call. (For functions defined without a prototype, parameters are converted inside the function definition.) A constraint requires that the conversion path exist. The implicit conversion rules, for assignment, simply specify that the value being assigned is converted to the object assigned to. There are constraints that ensure that this conversion is possible.

1011 function definition ends with ellipsis
999 argument type may be assigned
simple assignment

C90

The wording that specifies the use of the unqualified version of the parameter type was added by the response to DR #101.

C++

The C++ Standard specifies argument-passing in terms of initialization. For the types available in C, the effects are the same.

> *When a function is called, each parameter (8.3.5) shall be initialized (8.5, 12.8, 12.1) with its corresponding argument.*

5.2.2p4

Other Languages

This is how call by value is usually specified to work in languages that support some form of function prototypes.

Common Implementations

Passing an argument is not always the same as performing an assignment. In the case of arguments, the storage used to pass the value may be larger than the type being passed because of alignment requirements. For instance, arguments may always be passed in storage units that start at an even address. If characters occupy 8 bits, the unused portion of the parameter storage unit may need to be set to zero.

Coding Guidelines

The issues involved in any implicit conversion of arguments are the same as those that apply in other contexts.

operand convert automatically

1020 The ellipsis notation in a function prototype declarator causes argument type conversion to stop after the last declared parameter.

Commentary

Type conversion does not stop; as such, there are no available parameter types to convert to. However, the standard does specify another rule that may cause argument type conversions (see following C sentence).

C++

There is no concept of starting and stopping, as such, argument conversion in C++.

Other Languages

Common Lisp supports functions taking a variable number of arguments. However, it uses dynamic typing and it is not necessary to perform type checking and conversion during translation.

Common Implementations

Some implementations treat the arguments corresponding to the ellipsis notation differently from other arguments. The ellipsis notation makes it very difficult to specify, in advance, optimal passing conventions for those arguments. Some implementations put the arguments corresponding to the ellipsis notation in a dedicated block of storage that makes them easy to access using the va_* macros. Other implementations simply push these arguments onto the stack, along with all other arguments.

The default argument promotions are performed on trailing arguments. 1021

Commentary

default argument promotions 1009
called function no prototype 1008

The rationale for performing the default argument promotions on trailing arguments is the same as that for performing them on arguments of calls to functions where no prototype is visible. There is also the added benefit of simplifying the implementation of the va_* macros by permitting them to assume that these promotions have occurred; for instance, an implementation can rely on an object having type **short** having been converted to type **int** when passed as a trailing argument. These macros are specified to have undefined behavior if one of their arguments has a type that is not compatible with its promoted type.

Coding Guidelines

Some developers incorrectly assume that because a prototype is visible the arguments corresponding to the ellipsis notation are passed as is; that is, no implicit conversions are applied to them at the point of call (this assumption only leads to a fault is a certain combinations of events occur). Other than developer training, there is no obvious worthwhile guideline recommendation.

Example

```
extern unsigned char uc;

extern void f(int, char, float, ...);

void g(void)
{
f(1, uc, 1.2F, 99, 3.4F);
f(1, uc, 1.2F, 5.6, 'w', 0x61, 'y');
}
```

No other conversions are performed implicitly; 1022

Commentary

This C sentence specifies what happens when there is no function prototype in scope at the point of call.

C++

In C++ it is possible for definitions written by a developer to cause implicit conversions. Such conversions can be applied to function arguments. C source code, given as input to a C++ translator, cannot contain any such constructs.

Common Implementations

base document

The base document required that arguments having a structure or union type be converted to pointers to those types. This conversion was performed because the passing of arguments having structure or union types was not supported. It was assumed that, if an argument had such a type, a pointer to it had been intended and the translator implicitly took its address.

1023 in particular, the number and types of arguments are not compared with those of the parameters in a function definition that does not include a function prototype declarator.

Commentary

No comparison occurs even if the function definition, that does not use a prototype, is contained in the same source file as the call and occurs lexically before it in that file. The number and types of arguments are only compared with those of the parameters in a function definition if it uses a prototype and occurs lexically prior to the call in the same source file. In all other cases it is the visible declaration that controls what checks are made on the arguments in the call. The C Standard does not require any cross-translation unit checks against function definitions.

C++

All function definitions must include a function prototype in C++.

Other Languages

This behavior is common in languages that do not have a construct similar to a function prototype declarator. Without any parameter information to check against, there is little that implementations can do. The behavior for the case where a function call appears in the source textually before a definition of the function occurs varies between languages. Some (e.g., Algol 68) require that argument and corresponding parameter types be compared, which requires the implementation to operate in two passes, while others (e.g., Fortran) have no such requirement.

Common Implementations

Translators and static analysis tools sometimes perform checks on the arguments to functions that have been defined using the old-style definition. This checking can include the following:

- *Comparing the number and type of arguments against the corresponding parameters.* This can be done if the called function is defined in the same source file as the call, or by performing cross translation unit checks during translation phase 8.
- *Comparing the types of the arguments in different calls to the same function.* Often more than one call to the same function is made from the same source file. Flagging differences in argument types between these different calls provides a consistency check, although it will not detect differences between argument types and their corresponding parameter types.

1024 If the function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function, the behavior is undefined.

function call
not compatible
with definition

Commentary

pointer to function
converted

function
compatible types

For this situation to occur either a pointer-to function type has to be explicitly cast to an incompatible pointer to function type, or the declaration visible at the point of call is not compatible with the function definition it refers to. A cast could occur at the point of call, or when a function designator is assigned to a pointer-to function type.

Note: Calling a function, where the visible declaration does not include a prototype, with a particular set of argument types does not affect the type pointed to by the expression that denotes the called function (there is no creation of a composite type).

C++

In C++ is it possible for there to be multiple definitions of functions having the same name and different types. The process of selecting which function to call requires that a list of viable functions be created (it is possible that this selection process will not return any matching function).

It is possible that source developed using a C++ translator may contain pointer-to function conversions that happen to be permitted in C++, but have undefined behavior in C.

Common Implementations

Implementations usually treat such a call like any other. They generate machine code to evaluate the arguments and perform the function call. Unexpected behavior is likely to occur if the argument types differ from the corresponding parameter types and the two types have different sizes.

Differences in the return type can result in unexpected behavior, depending on the implementation technique used to handle function return values. In some implementations the return value, from a function, is passed back in a specific register. A difference in function return type can result in the value being passed back in a different register (or a storage location). Other implementations pass the return value back to the caller on the stack. A difference in the number of bytes actually occupied by the return value and the number of bytes it is expected to occupy can result in a corrupt stack (which often holds function return addresses) leading to completely unpredictable program behavior.

stack

Coding Guidelines

Intentionally calling a function defined with a type that is not compatible with the pointed-to function type, visible at the point of call, is at best relying on an implementation's undefined behavior being predictable (for argument-passing and values being returned). This usage may deliver different results if the source is translated using different implementations, or even different versions of the same implementation. If this usage occurs within existing code, it potentially ties any retranslation of that source to the translator originally used.

There is a programming technique that relies on the arguments in a function call depending on the context in which they occur. For instance, an array of pointers to function may have elements pointing at more than one function type.

```
extern int f0(), f2(), f4();
extern int f1(), f3();

int (*(a[5]))() = {f0, f1, f2, f3, f4};

void g(int p)
{
if (p % 2)
   a[p](1, 2);
else
   a[p](1);
}
```

ellipsis supplies no information
identifier ?? declared in one file
identifier ?? with extern linkage
reference shall #include

An alternative technique that does not rely on implementation-defined behavior is to declare the various f functions and the array element type using a function prototype that contains an ellipsis. Adhering to the guideline recommendations on defining an external declaration in one source file, which is visible at the point where its corresponding function is defined prevents a visible function declaration from being incompatible with its definition.

function call sequence point

The order of evaluation of the function designator, the actual arguments, and subexpressions within the actual arguments is unspecified, but there is a sequence point before the actual call. 1025

Commentary

return expression sequence point

The sequence point before the call guarantees that any side effects in the evaluation of the arguments, or the function designator, will have occurred prior to the call. When a value is returned, there is also a sequence point immediately before the called function returns.

C++

5.2.2p8

All side effects of argument expression evaluations take effect before the function is entered.

While the requirements in the C++ Standard might appear to be a subset of the requirements in the C Standard, they are effectively equivalent. The C Standard does not require that the evaluation of any other operands, which may occur in the expression containing the function call, have occurred prior to the call and the C++ Standard does not prohibit them from occurring.

Other Languages

A few languages do specify an evaluation order for the arguments in a function call; for instance, Java specifies a left-to-right order.

Common Implementations

Most implementations evaluate the arguments in the order that they are pushed onto the call stack, with the function designator evaluated last (this behavior usually minimizes the amount of temporary storage needed for argument evaluation). The most common order for pushing arguments onto the call stack implies a right-to-left evaluation order. Using this order makes it possible to pass variable numbers of arguments, the address of the last argument (the rightmost one in the argument list) in a call to a function may not be known when its definition is translated. However, the address of the first argument, in the argument list, can always be made the same relative to the frame pointer of the called function by pushing it last (see Figure 1000.1). By being pushed last the value of the first argument is always closest to the local storage, and successive arguments further way.

Coding Guidelines

This unspecified behavior is a special case of an issue covered by a guideline recommendation. Function calls are sometimes mapped to function-like macro invocations. In this case an argument that contains side effects may be evaluated more than once (in the body of the macro replacement). This issue is discussed elsewhere.

?? sequence points
all orderings give same value

macro
arguments separated by comma

Example

```
#include <stdio.h>

int glob = 0;

int f(int value, int *ptr_to_value)
{
if (value != *ptr_to_value)
   printf("first argument evaluated before assigned to glob\n");
if (value == *ptr_to_value)
   {
   printf("value == pointer to value\n");
   /*
    * Between the previous equality test and here glob may
    * have been assigned the value 1 (in main).  Let's check...
    */
   if (value != *ptr_to_value)
      printf("value != pointer to value\n");
   }
return 1;
}

int main(void)
{
(glob = 1) + f(glob, &glob);
}
```

1026 Recursive function calls shall be permitted, both directly and indirectly through any chain of other functions.

functi
rec

Commentary

object storage
outside function image

This is a requirement on the implementation. It is supported by a restriction on where storage for objects can be allocated. The standard does not specify any minimum limit on the depth of nested function calls that an implementation must support.

Other Languages

Some languages do not require implementations to support recursive function calls. Fortran (prior to Fortran 90) and some dialects of Basic do not support recursive calls (although some implementations of Fortran did allow recursion). Some languages (e.g., Fortran 90, and later PL/1 and CHILL) require that functions called recursively be declared using the keyword **recursive**.

Common Implementations

The implication of this requirement is that objects defined locally within each function definition must be allocated storage outside of the sequence of machine instructions implementing a function.

Recursive function calls are rare in most applications. Some implementations make use of this observation to improve the quality of generated machine code. The default behavior of the implementation being to assume that a program contains no recursive function calls. If a function is not called recursively, it is not necessary for the implementation to allocate storage for its local objects every time it is called— storage for these local objects could be allocated in a global area at a known, fixed address; these addresses can be calculated at link-time using the program's call tree to deduce which function lifetimes are mutually exclusive. Objects local to a set of functions whose lifetimes are mutually exclusive, can be overlaid with each other, minimizing the total amount of storage that is required.

Removing the need to support recursion removes the need to maintain a stack of return addresses. One alternative sometimes used is to store a function's return address within its executable code (ensuring that the surrounding instructions jump around this location).

Coding Guidelines

Many coding guideline documents ban recursive function calls on the basis that their use makes it impossible to calculate, statically, a program's maximum storage and execution requirements. (This is actually an overstatement; the analysis is simply very difficult and only recently formalized.[2, 3]) However, in some mathematical applications, recursion often occurs naturally and sometimes requires less effort to comprehend than a nonrecursive algorithm. Also, recursive algorithms are often easier to mathematically prove properties about than their equivalent iterative formulations.

Rather than provide a guideline recommendation and matching deviation, these coding guideline subsections leaves it to individual projects to handle this special case, if necessary.

EXAMPLE In the function call 1027

```
(*pf[f1()]) (f2(), f3() + f4())
```

the functions **f1**, **f2**, **f3**, and **f4** may be called in any order. All side effects have to be completed before the function pointed to by **pf[f1()]** is called.

Commentary

An order of execution could be guaranteed by using the comma operator:

```
(t4=f4(), t3=f3(), t2=f2(), t1=f1(),  (*pf[t1]) (t2, t3 + t4))
```

Common Implementations

Because its value is needed last, the function f1 is likely to be called last.

Forward references: function declarators (including prototypes) (6.7.5.3), function definitions (6.9.1), the **return** statement (6.8.6.4), simple assignment (6.5.16.1). 1028

References

1. A. Bauer. Compilation of functional programming languages using GCC - Tail calls. Thesis (m.s.), Munich University of Technology, Jan. 2003.
2. J. Blieberger. Real-time properties of indirect recursive procedures. *Information and Computation*, 171:156–182, 2001.
3. J. Blieberger and R. Lieger. Worst-case space and time complexity of recursive procedures. *Real-Time Systems*, 11(2):115–144, 1996.
4. B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4):313–351, 1995.
5. P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software–Practice and Experience*, 22(5):349–369, 1992.
6. B.-C. Cheng. *Compile-Time Memory Disambiguation for C Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 2000.
7. D. E. Corporation. *VAX11 780 Architecture Handbook*. Digital Equipment Corporation, 1977.
8. N. S. Coulter and N. H. Kelly. Computer instruction set usage by programmers: An empirical investigation. *Communications of the ACM*, 29(7):643–647, July 1986.
9. C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7^{th} USENIX Security Conference*, pages 63–78, Jan. 1998.
10. J. W. Davidson and D. B. Whalley. Methods for saving and restoring register values across function calls. *Software–Practice and Experience*, 21(2):459–472, Feb. 1991.
11. E. Eckstein and A. Krall. Minimizing cost of local variables access for DSP-processors. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES 99)*, volume 34.7 of *ACM SIGPLAN Notices*, pages 20–27. ACM Press, May 5 1999.
12. M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In 10^{th} *USENIX Security Symposium*, pages 55–66, Aug. 2001.
13. B. Furht. A RISC architecture with two-size, overlapping register windows. *IEEE Micro*, 8(2):67–80, Mar. 1988.
14. R. J. Hall. Call path refinement profiles. *IEEE Transactions on Software Engineering*, 21(6):481–496, June 1995.
15. IAR Systems. *PICmicro C Compiler: Programming Guide*, iccpic-1 edition, 1998.
16. IBM. *Developing PowerPC Embedded Application Binary (EABI) Compliant Programs*. IBM, Sept. 1998.
17. J. Ichbiah, J. Barnes, R. Firth, and M. Woodger. *Rationale for the Design of the Ada Programming Language*. Cambridge University Press, 1991.
18. T. Instruments. *TMS320C2x/C2xx/C5x Optimizing C Compiler User's Guide*. Texas Instruments, Inc, spru024e edition, Aug. 1999.
19. S. C. Johnson and D. M. Ritchie. The C language calling sequence. Technical Report Computing Science Technical Report No. 102, Bell Laboratories, Sept. 1981.
20. R. Leupers and F. David. A uniform optimization technique for offset assignment problems. In *Proceedings on 11^{th} international symposium on System synthesis*, pages 3–8, Dec. 1998.
21. W.-Y. Lin. An optimizing compiler for the TMS320C25 DSP processor. Thesis (m.s.), University of Toronto, Department of Electrical and Computer Engineering, 1995.
22. M. M. Martin, A. Roth, and C. N. Fischer. Exploiting dead value information. In *Proceedings of the 30^{th} Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'97)*, pages 125–135. IEEE, Dec. 1997.
23. Microchip. *PIC18CXX2 High-Performance Microcontrollers with 10-Bit A/D*, ds39026b edition, 1999.
24. A. Milanova, A. Rountev, and B. G. Ryder. Precise call graph construction in the presence of function pointers. In *Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)*, pages 155–162, Oct. 2002.
25. J. S. Miller and G. J. Rozas. Garbage collection is fast, but a stack is faster. Technical Report A.I. Memo No. 1462, M.I.T., Mar. 1994.
26. G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology*, 7(2):158–191, Apr. 1998.
27. G. C. Murphy, D. Notkin, and E. S. C. Lan. An empirical study of static call graph extractors. In *Proceedings of the 18^{th} International Conference on Software Engineering*, pages 90–99. IEEE Computer Society Press/ACM Press, 1996.
28. R. Muth, S. Debray, S. Watterson, and K. de Bosschere. alto: A link-time optimizer for the DEC Alpha. Technical Report TR98-14, The Department of Computer Science, University of Arizona, Wednesday, Dec. 9 1998.
29. PTSC. *Patriot C Development Tools Reference Manual*. Patriot Scientific Corporation, San Diego, CA, 1.1 edition, Feb. 2001.
30. M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlichting. System call clustering :A profile-directed optimization technique. www.cs.arizona.edu, May 2002.
31. R. Rakvic, E. Grochowski, B. Black, M. Annavaram, T. Diep, and J. P. Shen. Performance advantage of the register stack in Intel Itanium processors. In 2^{nd} *Workshop on Explicitly Parallel Instruction Computing Architecture and Compilers (EPIC-2)*, Nov. 2002.
32. A. Rao. Compiler optimizations for storage assignment on embedded DSPs. Thesis (m.s.), University of Cincinnati, Oct. 1998.
33. L. J. Shustek. *Analysis and performance of computer instruction sets*. PhD thesis, Stanford University, Jan. 1978.
34. The Santa Cruz Operation. *System V Application Binary Interface: MIPS RISC processor Supplement*. The Santa Cruz Operation, Inc, Santa Cruz, CA, USA, 3rd edition, Feb. 1996.
35. The Santa Cruz Operation. *System V Application Binary Interface: SPARC processor Supplement*. The Santa Cruz Operation, Inc, Santa Cruz, CA, USA, third edition, 1996.
36. The Santa Cruz Operation. *System V Application Binary Interface: Intel386 Architecture Processor Supplement*. The Santa Cruz Operation, Inc, Santa Cruz, CA, USA, fourth edition, Mar. 1997.
37. D. L. Weaver and T. Germond. *The SPARC Architecture Manual*. Prentice Hall, Inc, ninth edition, 2000.

38. C. A. Wiecek. A case study of VAX-11 instruction set usage for compiler execution. In *Proceedings of the first International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 177–184, Mar. 1982.

39. J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In 10^{th} *Network and Distributed System Security Symposium (NDSS'03)*, pages 149–162, Feb. 2003.

40. S. Zhang and B. G. Ryder. Complexity of single level function pointer aliasing analysis. Technical Report LCSR-TR-233, Rutgers University, Aug. 1994.

41. S. Zucker and K. Karhi. *System V Application Binary Interface: PowerPC processor Supplement*. SunSoft, Mountain View, CA, USA, 802-3334-10 edition, Sept. 1995.