Safe arrays and pointers for C

John Nagle

Discussion draft – August, 2012

Introduction

Buffer overflows continue to plague C and C++ programs. This is a draft proposal for dealing with that problem. The basis of this proposal is to define means for expressing the size of arrays in C. C already has fixed-size arrays with useful semantics. In this proposal, the existing syntax for fixed-size arrays is generalized to allow known-size arrays, where the size of the array is known at variable initialization time. With relatively minor and compatible changes to C, the most troublesome causes of program crashes and security vulnerabilities can be dealt with.

In any useful program, each array has a size known to the programmer. In C, there is currently no way to consistently express that size in the language. This proposal adds that capability.

Scope

An ISO document on common programming language vulnerabilities, "Information Technology — Programming Languages — Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use", ISO/IEC TR 24772, provides a standard taxonomy of flaws in programming language designs. The specific flaws in C addressed by this proposal are:

- 6.8 String Termination
- 6.9 Buffer Boundary Violation (Buffer Overflow)
- 6.10 Unchecked Array Indexing
- 6.11 Unchecked Array Copying
- 6.12 Pointer Casting and Pointer Type Changes
- 6.13 Pointer Arithmetic
- 6.14 Null Pointer Dereference
- 6.40 Type-breaking Reinterpretation of Data

These are the errors most closely associated with program crashes and low-level security vulnerabilities.

Not addressed are:

- 6.15 Dangling Reference to Heap
- 6.41 Memory Leak

Preventing errors of those types requires automatic memory management or extensive static analysis. Those flaws are difficult to fix without unacceptably large changes to the C language. Informally, a problem with adding garbage collection to C is that C is the language in which one writes the garbage collector.

Summary of language changes

The language changes required are minor.

Relaxation of existing restrictions

- Expressions allowed in most array size declarations. (This is the key enhancement.)

Features imported from C++11

- **auto**
- References

New reserved words

- **force_cast** *(storage class, for casts only, allows forcing certain casts in strict mode)*
- **void_space** *(type, for arrays of memory with unspecified contents)*

New predefined functions

- **size_t lengthof(***array***);**
- *arraytype* **&(array_slice(***array, start, end***))[***end - start***];** *(Generic function)*
- *typeofvalue* **init_space(***variable, value***)** *(Generic function)*
- **void clear_space(***variable***);** *(Generic function)*

New pragma

- `#pragma strict`

New restrictions (non-strict mode)

- None on existing code other than avoidance of the new reserved words.

New restrictions (strict mode)

- Conversion from a pointer to an array to a reference to an array of known length requires an explicit **force_cast**, to indicate that the programmer has commanded an unsafe operation.
- Conversion from anything which can be NULL to a reference implies a run-time **assert** check.

- Unions may not contain pointer or reference types, and all parts of a union must have the same length.
- To be used for pointer arithmetic, a pointer variable must be initialized at declaration to an element of an array.
- The only allowed values for the pointer variable used in pointer arithmetic are ones which point to an element of the array to which it was originally assigned. (Pointing to an element one past the end is allowed, for backwards compatibility.

Other

- The type of a string constant is a reference to an array, rather than a pointer to an array.

Rationale

Goals

- No change to run-time representation of data. (no descriptors or "fat pointers")
- Minimal modifications to the C language.
- Define "strict" and "non-strict" forms of C modules; existing C99 code can be compiled as "non-strict" modules and "strict" modules can interoperate with non-strict modules.
- If all modules are of "strict" form, and no unsafe overrides are used, programs should be protected against the flaws listed above.
- Hidden memory management mechanisms (garbage collection, reference counting) should not be required.

Previous work

- Microsoft source code annotation language (SAL)
- Cyclone (AT&T research language)
- SCC, the Safe C compiler.
- Ccured
- MemSafe
- Incompatible dialects and related languages

Fixed-size arrays in C and C++

The basis of this proposal is to define means for expressing the size of arrays in C. Most arrays in C and C++ are passed around as pointers, with no size information. There is, however, an exception – fixed sized arrays. C and C++ support fixed-sized arrays. Fixed-size arrays are typed objects of known length. They can be defined as types, and, in C++, references to fixed size arrays are fully supported.

```
int arr100[100];   // an array of 100 ints
arr100& arr100r;   // ref to array of 100 ints
```

This is valid C++ today. C++ already supports the concept of a reference to an array. (Arrays of references are prohibited by the C++ standard, but references to arrays are not.)

Where the compiler knows the size of an array, we can use that information. It can be accessed via **sizeof**. This is convenient, but only useful for arrays of fixed size.

C99 dynamically sized arrays

C99 added support for dynamically sized temporary arrays. This feature is often used in numerical code, where it is common to call a function which needs a temporary array for the duration of the function call.

```
float mathfn(const float in[ ], size_t n)
{    float workarr[n];   // work array of length n
     …
     return(result);
}
```

This was the first place in C where an array could be resized at compile time with the compiler aware of the size.

Known-size arrays.

The existing syntax and semantics of fixed-size arrays, C99 dynamically sized arrays, and C++ references can, with slight extensions, allow the support of known-sized arrays. The only syntactical change required is to allow expressions in a few places where, at present, only constants are allowed. It is proposed to allow expressions in array sizes in type declarations in the following places:

- local variable and typedef declarations
- function parameter declarations
- casts within initialization expressions
- structure definitions (under certain restrictions)

When a known-sized array is defined or passed as a parameter, its size is the value of the dimension expression evaluated at initialization time.

This allows a function such as the following:

```
void copybyref(int (&a)[n], const int (&b)[n], size_t n)
{
    for (int i = 0; i < lengthof(a); i++)
    {   a[i] = b[i]; }
}
```

This is a size-safe copy, intended to be used as follows:

```
int a0[100], a1[100];    // defined with a type declaration
copybyref(a1, a0, 100);  // copy array of known size

int b0[50], b1[50];      // smaller arrays
copybyref(b0, b1, 50);   // copy array of a different size
copybyref(b0, a0, 100);  // size mismatch - compile time error
```

The last "copybyref" call is an error, because the array sizes do not match. For a constant "n" this can be caught at compile time. For a variable n, it has to be a run time check.
The compiler must check, at a call to "copybyref":

```
assert(lengthof(a) == n);
assert(lengthof(b) == n);
```

If the programmer calls "copybyref" as follows:

```
copybyref(b0, a0, lengthof(b0));
```

The checks become

```
assert(lengthof(b0) == lengthof(b0));
assert(lengthof(a0) == lengthof(b0));
```

The first check is optimized out as an identity. The second check still requires a run-time check. But only one check per function call is required. This is far cheaper than general subscript checking.

Here, a run-time check is required only because two arrays must match. In many common cases, especially some of the classic causes of buffer overflows, there is no additional overhead. For example, the classic UNIX *read* call would now be expressed as:

```
int read(int fd, void_space(&buf)[n], size_t n);
```

(The type `void_space` is discussed below; it's simply a type with a size of 1, usable for arrays, for which the bytes have no predetermined meaning. Think of it as **void*** with length.)

The required check to be generated by the compiler is

```
assert(lengthof(buf) == n);
```

If called with

```
int buf[512];
...
int stat = read(fd, buf, lengthof(buf));
```

the check required is

```
assert(lengthof(buf) == lengthof(buf));
```

which is optimized out as an identity.

Worth noting is that the function declaration of **read** above is, in non-strict mode, compatible with existing code which passes **buf** as a pointer to a char.

Why references?

A key part of this proposal is the use of C++ style references in C. As shown above, references to arrays can already contain size information about the array. Pointers to arrays in C refer to the type of the first element, with no array length information. Changing the semantics of pointers in C would break existing code. Adding references to the C language will not break anything. C++ already has both references and pointers. The relationship between the two is well understood and not a cause of trouble, so this can be done with confidence.

C99 already allows variable declarations to be interspersed with executable code. References must be initialized in C++, and this restriction would be maintained with references in C.   This proposal allows expressions in array sizes in declarations, and those expressions must be evaluated where the declaration appears.

```
int b[100];                         // array of 100 ints
...
int (&rb)[lengthof(b)] = b;         // ref to array b
```

Assignments to references are subject to type checking. Where the reference on the left hand side has a known length determined at run time, this may imply a run-time assert check on array size. This is considered a subscript check, as defined below.

Strict mode

The enhancements to C syntax and semantics described here are backwards compatible with existing C code. Beyond this is "strict mode" which disallows some unsafe constructs and operations. Strict mode requires known size arrays in situations where overflow is possible. "Strict mode" is defined on a per-source-file basis. Strict and non-strict code can interoperate.

New rules enforced in strict mode:

- Conversion from a pointer to an array to a reference to an array of known length requires an explicit **force_cast**, to indicate that the programmer has commanded an unsafe operation.
- Conversion from anything which can be NULL to a reference implies a run-time **assert** check.
- Unions cannot contain pointers or references, and all alternatives of a union must have the same size.

Programs written in strict mode should be protected against the list of memory safety errors covered in the "scope" section above. This should eliminate most buffer overflow problems.

Strict mode for a compilation unit is indicated by

```
_Pragma("strict")
```

or

```
#pragma strict
```

Implications of known-size arrays

With known-sized arrays and strict mode introduced, their use and consequences can be examined.

How array size information is stored

No array descriptors are generated by the compiler. The programmer tells the compiler the size of the array as an expression, and that expression is evaluated at the point in the program where the relevant declaration appears.

```
int asize = 100;    // size wanted
int atab[asize];    // declare known-size array
asize = 200;        // this does not resize the array
printf("Length of atab is %d\n", lengthof(atab)); // prints 100
```

In the above worst case example, the compiler must generate a temporary is needed to store the size of the array, because the inputs to the expression defining its size change. This is usually unnecessary.

A struct with an array as the last component

An especially useful idiom is a structure with an array as the last component. C supports this now, but the size of the array is always undefined, expressed with **[ ]** . This proposal allows such arrays to be of known size. The array, and the structure, then have a known size. and

A simple example is

```
typedef struct msgitem {
     const size_t len;
     char itemvalue[len];
};

struct msgitem firstmsg = { 100, 0 }; // empty msgitem, size 100
```

The length, **len**, is an element in the same structure. This ties the length of the array to a value which can be found from the structure. This tie is a property of the type declaration. The general rule for such declarations is that the dimension expression is evaluated in the context of the struct. Fields of the struct may appear in the expression. Such structs should be initialized as a unit, with an initialization expression.

For such types, the size of the type, and the length of the array, are known to the language. They can be accessed with **sizeof** and **lengthof**. If subscript checking is enabled, lengths are checked on assignments to and from the entire variable.

The examples in Appendix 2 show a safe string type implemented using this approach.

Associating length with an array pointer in a structure

Alternatively, a user might want a structure which carries both the array and its size. The pointer-based form is

```
typedef str { size_t len, char* data};
```

New reference form:

```
typedef str { size_t len, char (&data)[len]);
…
str s1 = {n, MALLOC(char, n) };    // structure initialization
```

The key point here is that `char (&data)[len]` is a reference to a string of known length. As in the previous example, **len**, is an element in the same structure, and ties the length of the array to a value which can be found from the structure. Multiple arrays may be present in the same structure. However, all such arrays must be allocated during structure initialization, because a reference cannot be null.

Array size expressions

Array size expressions in declarations of arrays of known size present some special problems. As expressions in declarations, they have unusual scope restrictions. In C99, expressions are allowed in array declaration sizes provided that all variables referenced are compile-time constants, and the only functions allowed in such expressions are built-ins such as **sizeof**. For this proposal, that restriction is relaxed as follows:

- In function declarations, any formal parameter may be used in a size expression, provided that this does not result in a dependency loop.
- In struct declarations, any structure field name may be used in a size expression, provided that this does not result in a dependency loop.

Array size expressions are thus quite limited in form. This is necessary, because they must produce a consistent result regardless of the scope in which they are evaluated. In particular, the array size expressions evaluated where a function is called must produce the same results when recomputed inside the function when the function is entered.

It is specifically allowed to use a formal parameter in a size expression which appears before the declaration of that parameter. This allows classic idioms such as

```
int read(int fd, char (&buf)[n], size_t n);
```

where the size appears after the array being sized. While this requires extra work in parsing, it eliminates the need to make incompatible changes to interfaces that have been standardized for decades.

Type void_space

C uses the idiom **void *** for a pointer to data of undetermined type. Conversion to and from void without coercion is allowed. However, an array of **void** is not allowed; **void** has no size. A way is needed to talk about arrays of space without implying type. So it is proposed to add the built-in type **void_space.**

A variable of **void_space** is essentially the machine allocation unit of storage. This is a byte on almost all current hardware. (9 bit quarter words on Unisys ClearPath 36-bit mainframes.) The size of void_space is 1. An array of **void_space** is an array of bytes. Any variable, array, or struct that does not contain pointers or references can be converted to or from an array of **void_space**, without a cast, if the sizes match. A reference to an array of **void_space** can be converted to **void ***, and in non-strict code, the reverse is allowed.

Thus, any function which now uses a parameter of **void*** to represent an array can instead use a reference to an array of **void_space**. For example,

```
int read(int fd, void_space (&)buf[n], size_t n);
```

defines the standard POSIX/Unix read function in a way which is size-safe when called from strict code, and callable from non-strict code. This preserves programmer expectations about these common operations while enforcing size safety in strict code.

Type **void_space** values are converted to type **unsigned int** if used in an arithmetic context.

Casts

The syntax of casts follows that for declarations. As with reference declarations, the size must be evaluated at the point of the cast. Casts with expressions in array sizes are allowed in initialization expressions.

Ordinary casts must be memory-safe in strict mode. Unsafe casts must be expressed using some specific syntax, as with **reinterpret_cast** in C++. Unfortunately, the syntax of **reinterpret_cast** uses the corner-bracket template notation of C++, and is inappropriate for C. Thus, a syntax compatible with C is needed.

This, like most syntax issues, is a politically touchy problem. A solution should be

- Consistent with the expectation of C programmers.
- Backwards compatible with existing code
- Findable by simple searches to enable easy code auditing.

One option is a special storage class, used only in casts. A phrase such as **forced_cast** might be appropriate:

```
int val = 1234;
char* ptr = (forced_cast char*)val;     // UNSAFE
```

Even in strict mode, this would be required only when forcing a conversion which could impact memory safety. So such casts would be quite rare.

A few casts can generate run-time checks. Conversion of a pointer to a reference must generate an **assert** check. Such checks are enabled when subscript checking is enabled.

Dynamic allocation ("malloc" and its friends)

This is one of the more difficult problems to handle safely. Yet, as it is frequently a source of program bugs of the buffer overflow variety, it it necessary to do so. The problems which must be addressed include:

- The classical return type of **malloc**, "void *", cannot express size information.
- The return type from memory allocation must be convertible to other types.
- Because **malloc** can return NULL, it is not permissible to directly generate a reference from **malloc.**

- Pointers to references are not allowed.
- Structures which contain references must be initialized.
- The syntax for allocating memory must be concise.
- Exceptions are not available in C.
- Backwards compatibility with existing code must be maintained in non-strict mode.

All those constraints, taken together, create a tough problem. A few options, which should be viewed as illustrative, not part of a proposed library:

- Allocation functions which raise a signal on failure. This impacts program structure, but provides the cleanest syntax.

  ```
  void_space (&smalloc(size_t len))[len];
  ```

  but there's no way to handle the error condition in in-line code. It's worth providing this for routine applications which do not need sophisticated out-of-memory handling.

- Allocation functions with a failure callback. This is for programs which need to explicitly handle out of memory conditions

  ```
  void_space (&smalloc_failcheck(
       size_t len,
       void (*outofmemfn)(size_t)))[len];
  ```

  The *outofmemfn* function, if non-null, would be called on an out of memory condition. If it is able to relieve the memory shortage and return, the program can continue. If not, the function should escape the call, using a **longjmp***,* or terminate the program after any necessary closeout actions.

- Return a pointer to a structure which contains a reference, or return NULL. The string type example in the appendix uses this approach.

Initialization of dynamically allocated space

In C, "malloc" normally provides memory space containing old semi-random data. For most data types, this is not a safety problem. However, for pointer safety, we must insure that allocated items with pointers or references are initialized. Zeroing new memory space is not an option for references, and because we're encouraging the use of references in place of pointers, this has to be done right.

So, when a new struct or array or structs is allocated, and the struct type contains pointers or references,   it must be initialized from an valid initial value of that type. It would be preferable to do this without casting.

As C lacks generic functions other than built-ins, a type-safe way to do this is needed. A built-in generic function, **init_space,** is proposed. This is a size and type safe replacement for the classic "memcpy".

typeofvalue **init_space(***variable, value***)**

This simply copies *value* into *variable,* with type checking. The *variable* parameter can be an array of known or fixed size. This is needed because array assignment isn't allowed, and assignment to an array reference assigns the reference, not the content. In C++, we could write this as a template function, but in C, any generic function must be a built-in.

Example usage, where a structure with some references is to be initialized, is

```
struct bufferpair {
     size_t len;
     char (&) buf1[len];
     char (&) buf2[len];
     };
…
//   allocbufferpair – allocate a buffer pair
struct bufferpair* allocbufferpair(size_t len)
{    const itemproto bp = {len, smalloc(len), smalloc(len)};
     bufferpair& p = init_space(smalloc(sizeof(itemproto)),
          itemproto);
     return(bufferpair);
}
```

A C initializer is used to construct the object, which is then copied to the newly allocated space. No casts are required, yet this is both type and size safe. The **smalloc** function is from the previous example.

As a convenience, a safe replacement for the common "bzero" is provided:

void clear_space(*variable***)**

This simply sets the array to binary zero. It is a compile-time error to use this on an array of structures containing references.

Impact on other language features

The basic proposal has now been described. The impact of the proposal on various features of the C language is next.

Unions

In strict mode, elements of unions are restricted to non-pointer and non-reference types. Unlike struct types, union types may not have arrays of known but variable length.

In most existing code, unions are used primarily for data being passed through networks, files, and message passing systems. Such data, since it comes from external sources, seldom contains pointers. Adding language support for safe discriminated variant records, as was done in Cyclone, seems excessive for the infrequency with which it would be used.

In strict mode, the only way to override pointer type safety is through the explicit use of **force_cast.** This make it easy to search and audit programs for pointer type problems.

Multidimensional arrays

Multidimensional arrays of known size are supported. This simply generalizes the existing rules for multidimensional fixed-size arrays.

```
	size_t order = 3;
	…
	float tab[order][order]; // square array of order N.
```

As with arrays of fixed size, these are not arrays of pointers, but dense arrays of elements. This will be a great convenience in numerical work. There is also a performance gain. On modern CPUs, multiplying by row size to compute a row offset is faster than accessing a table of pointers.

Pointer arithmetic

Pointer arithmetic is a standard idiom of C programming. Although it would be safer to eliminate it from the language, that isn't feasible for an update to C.

The most common uses of pointer arithmetic use local variables which are bound to a single array throughout their life. Such pointers behave much like C++ iterators. So we can allow pointer arithmetic under much the same restrictions applied to iterators. These are:

- The pointer variable must be initialized at its declaration to an element of an array.
- The only allowed values for the pointer variable are ones which point to an element of the array to which it was originally assigned. (Pointing to an element one past the end is allowed, for backwards compatibility.

Pointers with such properties are safe for pointer arithmetic purposes, provided there is range checking. Range checking for such pointers is possible without "fat pointers", because the array to which they are bound is known at compile time.

Unsafe pointer arithmetic is allowed only through **force_cast.**

This allows a strict mode version of the classic C idiom:

```
	void bcopy(char (&dst)[len], char (&src)[len], size_t len)
	{    while (len--) { *dst++ = *src++; } }
```

Called from strict mode, the lengths of the parameters must match **len**, and the function itself has enough information to allow run-time subscript checking. Called from non-strict mode, this is compatible with the classic **bcopy** function.

If this were written with a **for** loop, subscript checks could be hoisted out of the loop, recognized as an identity, and eliminated, resulting in safe code without run time checks within the loop. On some platforms, one `mov` instruction could perform the copy without loss of memory safety.

String constants

String constants will now have type **char (&)[*length*] or wchar_t(&)[*length*]**. These types will convert to **char*** or **wchar_t*** for compatibility with non-strict code. The **sizeof** and **lengthof** functions can be applied and count the usual trailing null.

Null-terminated strings are a problem. A workable solution is to allow read access to const arrays without subscript checking. This allows reading junk, but not overwriting other variables.

Standard library safety

Library functions can be categorized as follows:

- Can be made fully compatible and safe with arrays of known size – **read**, **write**, **fread**, **fwrite**, etc.
- Unsafe and already deprecated – **gets**, **sprintf**, etc. These would be unavailable in strict mode.
- Safe for write, but not for read – **printf**, **snprintf**, etc. Permitted in strict mode on the grounds that the impact of disallowing them is too high.
- Unsafe – **sscanf,** which stores into memory based on format strings. Unavailable in strict mode.

Library issues are beyond the scope of this document, but appear to be manageable.

Subscript checking

With enough information available to perform subscript checking, implementations may offer it as an option. The question is what to do when a run time error is detected. The available options are to take the action defined for an “assert” failure, or to raise a signal. That decision can be left to the implementation.

It is strongly recommended that implementations which do subscript checking optimize it. As a minimum, the following is suggested:

- common subexpression optimization should be applied to subscript checks within loops.
- Within “for” loops, checks should be hoisted to the beginning of the loop and done once per loop if at all possible.
- Hoisted checks which resolve to constant expressions or identities should be evaluated at compile time, and reported as errors (if failing) or optimize out (if succeeding).
- The array must not have a scope narrower than that of the pointer variable, so that the array cannot go out of scope while the pointer variable is still in scope.

The goal of the above is to optimize out most, if not all, subscript checks in inner loops of common matrix calculations.

It is explicitly permitted to detect and act on a subscript error as soon as it becomes inevitable. In other words, if an implementation can detect that a subscript error will occur on some iteration of a loop, and there is no way to exit the loop before reaching that iteration, the subscript error can be reported at entry to the loop. A potential exit from the loop via “break” or “return” must inhibit this optimization. A potential exit via “longjmp” or “exit”, which might be hidden in a called function, would not be considered to inhibit this optimization.

Convenience features

With the language aware of array length, some convenience features can be added to make programming easier. These are all optional from the programmer's perspective, and need not be retrofitted to existing code.

“auto” storage type as in C++ 20xx.

“auto”, from C++11, is added to C. The motivation for this is the long declarations required for references to complex arrays.

“lengthof”

C has a built-in “sizeof” function. It's common to define “lengthof”, returning the number of elements of an array, as a macro. It is proposed to make “lengthof” a standard. Both “sizeof” and “lengthof” are defined for arrays of known size. This makes the size of an array immediately and reliably available to code.

Iteration over an array

Almost all languages defined in the last 20 years (with the notable exception of JavaScript) have syntax for iterating over all the elements of a collection. For arrays of known size, the compiler now has enough information to provide such a construct. The syntax of this may be controversial, and needs discussion. Some options include

```
for (auto x in arr) { … }

for (auto x forall arr) { … }
```

Making “in” a reserved word impacts existing code, but yields the nicest syntax. Using this feature with “auto” leads to concise loop statements which are not error prone. At this time, no specific proposal is made in this area. Suggestions are welcome.

Subarray syntax

At times, it is desirable to talk about a subarray of an array. The existing C syntax for this is

```
char arr[100]

char p[] = &arr[20];
```

This tends to be error-prone, as the size of the subarray is rather vague. So it is proposed to borrow the *slice* operation from Javascript:

```
char arr[100]

char (&p)[80] = array_slice(arr, 20, 100);
```

Following the Javascript and Python convention, the limits of slice are the start index and the end index + 1. Negative, overlapping, and out of bounds indices are errors and should be detected if subscript checking is enabled.

Another option is to borrow Python's subarray syntax:

```
Char arr[100];

char (&p)[80] = arr[20:100];  // reference to a subarray of arr
```

This is concise, but more controversial, as it involves new syntax. It has the advantage that subarray references could be allowed on the left side of an assignment.

Conclusion

This is a workable solution to a decades-old practical problem, a problem which has probably caused more computer program crashes than anything else in the history of computing.

Alternatives

Making C a memory safe language without unduly changing the language is difficult. This proposal is intended as a direction for standard C, and as such, must be quite conservative in the changes proposed.

Annotation systems, such as Microsoft's SAL, can be effective if used. Because they're not integrated into the language, they tend to make programs more verbose. Thus, they're rarely used without strong managerial pressure.

Cyclone has a reasonable set of extensions, but strays far enough from C that it is a different language. Cyclone offers three types of pointers - “normal”, “never-null”, and “fat”. New symbols (“@” and “?”) are used to designate the different types of pointers. “Fat” pointers imply hidden run-time machinery. This is a valid approach, but it is not C.

SCC, the Safe C compiler, is claimed to be able to detect all pointer and access errors at run time. However, it required using an elaborate data structure for every pointer and substantial run-time overhead.

C++ addresses safety by trying to hide the unsafe constructs of C under a layer of standard templates. In practice, the templated constructs tend to “leak” raw C pointers, and buffer overflows in C++ applications are still common.

Other variants such as Objective-C and Java are useful languages, but they are not C.

Limitations

This approach still doesn't solve the dangling-pointer problem. There's no good way to do that without more memory management machinery behind the scenes. The goal here was to avoid adding hidden machinery.

Appendix 1 – Examples

From the UNIX/POSIX API:

Original function declarations:

```
     int read(int fd, char* buf, size_t n);
```

New form:

```
     int read(int fd, void_space (&)buf[n], size_t n);
```

This is size-safe when called from new code, and compatible with old code that passes a "char*" as a parameter.

From Numerical Recipes in C, Gauss-Jordan elimination:

Original function declaration:

```
     void gaussj(float **a, int n, float **b, int m);
```

New form:

```
     void gaussj(float (a&)[n][n], int n, float (&b)[n][m], int m);
```

In the original, the array bounds have to be explained in comments.

(More to follow)

Appendix 2 – an illustrative string library

This is an illustration of how a safe string library, usable in strict mode, might be constructed. It is not part of the proposal to specify such a library at this time.

```
//
//  sstring -- strict string
//
//  A length, and a buffer.  The content is null-terminated and
//  the length is the buffer size.
//
struct sstring {
     size_t len;
     char s[len];
     };
```

```
//
//  sstring_alloc --  Safe string from null-terminated string
//
struct sstring* sstring_alloc(size_t len)
{
     //   Create prototype of dummy string
     struct sstring proto = { len, {0}};
     struct sstring* dest = // use prototype to calculate size
          (struct sstring*) malloc(sizeof(proto));
     if (dest == NULL) return(NULL);    // fails on out of memory
     *dest = proto;            // int from prototype
     return(dest);             // return pointer to string struct
}


//
//  sstring_make --  Safe string from null-terminated string
//
struct sstring* sstring_make(const char* src)
{
     const size_t srccnt = strlen(src); // copy size, including null
     //   Create prototype of dummy string
     struct sstring proto = { srccnt+1, {0}};
     struct sstring* dest = // use prototype to calculate size
          (struct sstring*) malloc(sizeof(proto));
     if (dest == NULL) return(NULL);    // fails on out of memory
     dest->len = proto.len;             // copy length
     //   Copy contents, including trailing null
     for (size_t i=0; i <= dest->len; i++) dest->s[i] = src[i];
     return(dest);             // return pointer to string struct

}

//
//  sstring_cat  - concatenate safe string
//
struct sstring (&sstring_cat(sstring &dest, const sstring &src)
{    const size_t destcnt = strlen(dest.s); // content length in dest
     const size_t srccnt = strlen(src.s);   // content length in src
     assert(destcnt + srccnt < dest.len);   // will it fit?
     for (size_t i = 0; i <= srccnt; i++)   // copy contents w/null
     {   dest.s[i+destcnt] = src.s[i];  }
     return(dest);      // return pointer to string struct
}

//
//  sstring_del - delete string
```

```
//
void sstring_delete(sstring* dest)
{    if (!dest) return;       // if already deleted, done
     free(dest);              // free string
}

//
//   demo  - strict string test demo
//
void demo(int fd)
{    sstring* s1 = sstring_make(“Hello”);
     sstring* s2 = sstring_make(“ ”);
     sstring* s3 = sstring_make(“World”);
     sstring* wrk = sstring_alloc(25);  // create a working string
     sstring_cat(wrk,s1);     // add some words
     sstring_cat(wrk,s2);
     sstring_cat(wrk,s3);
     //   Write the string out.
     //   Type checking insures the sizes match.
     //   “sizeof” is meaningful for this type.
     int stat = write(fd, wrk, sizeof(*wrk));
     //   Print the string using printf.
     //   No size checking here, but this is a const usage.
     printf(“%s\n”, wrk->s);
}
```