address is truncated by an assignment of a `T*` into a `U*` (such as on a word-addressed machine), the truncated address still refers to the original collected object.

Most, perhaps all, implementations of `new` and `malloc` already behave this way. Though the *ARM* is silent on the issue, returning maximally aligned storage is required to implement overloading of `operator new`.

A stronger notion of type validity would require that a variable of type `T*` contains a pointer to a value of type `T` or a class derived from `T`. We know of only one garbage-collection technique that benefits from strong type validity. This technique uses the declared types of pointers to determine the types of objects in the heap rather than tagging the objects themselves, saving the space overhead of the tags. But this technique doesn't fully extend to languages with class inheritance—objects of class types still need to be tagged. As far as we know, this technique has never been implemented, and its space savings for typical C++ programs would be negligible.

Further, like pointer validity, strong type validity would restrict compatibility with existing C++ code.

7. Safe subset

The safe subset is just that: a true subset of the C++ language [Ellis 91], requiring no extensions or changes. Programmers are assured that code written in the safe subset is GC-safe, that is, it follows the safe-use rules of the garbage collector. More importantly, they are assured that code written in the safe subset cannot be responsible for storage-related bugs caused by dangling pointers or references, memory smashes, null-pointer dereferences, or invalid array indices. Programmers mark safe code with a pragma, and the compiler ensures that such code uses only the safe subset. It also generates some run-time checks to ensure safe use of particular language features—a program attempting to use an invalid pointer will halt with an error. Code written in the safe subset can be ported without change to C++ implementations that don't provide the subset.

Programmers need not write in the safe subset to use garbage collection, but then the responsibility is on them to ensure GC-safety and to avoid storage bugs. Experience with other garbage-collected languages shows that automatic-enforcement of the safe-use rules is quite important, since collectors tend to scramble the heap arbitrarily when the rules are violated. Even when some parts of a program can't be written in the safe subset, using it in the rest of the program reduces the potential for mistakes—when tracking down storage bugs, the programmer can safely rule out all code written in the subset.

The run-time checks are designed to have fairly low overhead so that they can be used throughout development and even in production. However, programmers can disable them at any time, trading safety for efficiency.

The safe subset ensures portability among different implementations of garbage collection by enforcing implementation-independent safe-use rules. An unsafe program may work with some collectors but not others; for example, an unsafe program may work with a mark-and-sweep collector but break with a copying collector.

The design of the safe subset is based on long experience with languages like Cedar, Modula-2+, Modula-3, and Ada [Rovner 85a, Rovner 85b, Nelson 91]. The safe subsets of these languages are expressive enough for applications and all but the lowest-level systems code and run-time facilities (such as device drivers). This C++ safe subset is noticeably less restrictive than the subsets of those languages, however. In particular, a non-collected object can point at a collected object, a pointer may address the interior of another object, and pointers may be freely passed to libraries written in other languages.

7.1. Subset summary

The subset enforces safety by:

enforcing at run-time what's already illegal in C++ but current implementations don't prevent,

replacing built-in arrays with safe arrays provided by standard template classes,

preventing fabrication of invalid pointers, and

preventing dangling pointers and references.

To prevent dangling addresses of automatic variables, the subset prohibits pointers to automatic variables and storing references to automatic variables into heap objects.

A pragma, `#pragma safe`, declares source files and blocks as safe. The compiler ensures that only safe constructs are used in safe files and blocks.

The following features are disallowed in safe code by the compiler at compile-time:

pointer arithmetic
array subscripting
converting arrays to pointers

passing arguments to formal parameters of type "array of `T`" (`T[]` or `T[c]`) unless the formal is of type `T(&)[c]` ("reference to array `[c]` of `T`") or the argument is a string literal
all casts to types containing pointers, references, and functions, except widening casts and checked narrowing casts
union types containing pointers, references, or functions
overloading `operator new`
uninitialized pointer variables and members
`delete` and explicit calls to destructors
functions declared with ellipsis

The compiler generates run-time checks in safe code for the following constructs:

dereferencing a null pointer
applying `&` to an lvalue referencing an automatic variable
initializing a reference in an object created by `new` to an automatic variable
explicit use of `this` when it points to an automatic object, unless it is the operand of `*` or `->`
returning a dangling reference to an automatic variable
exiting a value-returning function without an explicit `return` or `throw`

If a check fails, a *checked run-time error* occurs and the program halts in an implementation-dependent way.

Two restrictions are placed on the compiler itself:

The storage for a temporary object used in the initialization of a reference must not be reused until the block defining the reference exits.

Arrays of pointers must be initialized to null.

Code-generator safety imposes further restrictions on the compiler. But code-generator safety is required regardless of whether the safe subset is used or not (see section 11).

Three standard classes, `Array`, `DynArray`, and `SubArray`, provide safe arrays with run-time subscript checks. The standard class `Text` provides efficient immutable strings.

In what follows, each of the restrictions is specified more precisely and discussed in detail. Everywhere an operator is mentioned, we mean the built-in operator, not any overloadings—there are no specific restrictions on overloaded operators.

The following terms are used by the subset definition. A type `T` is *pointer-containing* if:

`T` is a pointer type;
`T` is an array type whose element type is pointer-containing;
`T` is a class containing or inheriting a member whose type is pointer-containing; or
`T` is a union containing a pointer-containing type.

The definitions of *reference-containing* and *function-containing* are similar.

7.2. Pragma safe

☐ Programmers use pragmas to specify which code is written in the safe subset, and the compiler enforces the use of the safe subset in such code. Declarations, blocks, and parenthesized expressions can be specified safe or unsafe using the pragmas

```
#pragma safe
#pragma unsafe
```

The pragmas can occur syntactically anywhere a declaration can occur (in a file, class, or block) or at the beginning of a parenthesized expression:

```
(
#pragma unsafe
...)
```

The scope of the pragma extends to the end of the file, class, block, or expression containing the pragma or until the next `safe` or `unsafe` pragma, whichever comes first. There is an implicit `#pragma unsafe` at the beginning of every file. The scope of a pragma in an included header file ends at the end of the header file.

A name declared in the scope of a `safe` pragma is marked safe; all other names are marked unsafe.

It is a compile-time error for declarations, statements, and expressions in the scope of a `safe` pragma to use unsafe names or constructs prohibited by the safe subset.

The declaration of a function may be declared safe, while its definition is declared unsafe. In this case, the programmer is asserting that he believes the function cannot violate safety, even though it may be implemented with unsafe constructs. It is a programmer bug if the function ever violates safety when called from safe scopes.

A programmer is assured that a storage bug couldn't be caused by safe files and blocks, since the execution of a safe block cannot cause GC-safety to be violated. If a name's declaration and definition are both declared safe, then any use of the name in safe scopes will not violate GC-safety.

A class may declare both safe and unsafe member functions:

```
#pragma safe
class T {
    void f();
    #pragma unsafe
    void g(); }:
```

This allows programmers to provide unsafe methods that aren't intended for general, safe use.

As a rule, it is good to have as many safe interfaces as possible, even if some of their implementations are written in the unsafe language. An interface declared safe signals to clients that use of the interface shouldn't cause storage bugs. If the interface's implementation is also declared safe, then the compiler ensures that its use from safe scopes won't violate safety; but if the implementation is declared unsafe, then the programmer is merely asserting that its use won't violate safety, and it is a bug in the implementation if it does. If most of an interface or implementation is safe, it is best to declare the entire file as safe, using `#pragma unsafe` to identify the few unsafe declarations and blocks.

7.3. Enforcing what's already illegal

☐ A run-time check ensures that dereferenced pointer expressions are non-null.

Dereferencing a null pointer would violate the definition of safety. In most implementations, an unchecked dereference of null could result in garbage values.

In many, if not most, implementations, run-time checks for null are almost free—the first *n* bytes of an address space are unmapped, and attempts to reference them cause a virtual-memory exception. Only dereferences involving an offset larger than *n* or an offset unknown at compile time require explicit checks. For example:

```
struct S {char a[n]; int i;};
S* s = ...;
int j = s->i;
```

If `s` is null, then the dereference `s->i` won't cause a virtual-memory exception, since the word at address *n* isn't unmapped. The check requires a single instruction: before dereferencing `s`, it loads the word at offset 0 from `s` and discards the result.

- A run-time check ensures that a function never returns a reference to one of its automatic variables.

If a function returns a reference to one of its automatic variables, the reference will dangle as soon as the function's stack frame is popped. References can be returned by `return` and `throw` statements, either as the returned or thrown value, or as a member or element of such a value.

Checking returned references at run-time is cheap. On the MIPS R3000, for example, the following sequence checks that an address in register `$r` doesn't point to the current stack frame:

```
subu  $t, $r, $sp
sltiu $t, $t, framesize
bne   $t, 0, error
```

The check can be omitted wherever the compiler can determine statically that the returned reference was or was not initialized to an lvalue of a local automatic variable. For example, a reference-valued formal parameter or a pointer dereference couldn't possibly refer to a local automatic variable (in the safe subset). In general, the only returned references needing run-time checks are those whose initialization is not statically apparent, that occurs after entry into the function, and that might refer to local variables; such references can occur only as the result of function calls with reference parameters or of catching exceptions. For example:

```
int& f(int& i);
int& g() {
    int i = ...;
    return f(i);}
```

The initialization of the return value of `f(i)` is not statically apparent, so the compiler must generate a run-time check for `g`'s return. Note that references returned by inlined functions need not be checked (assuming the storage for the function's automatic variables persists until the callee exits).

Most returned references can be statically analyzed by the compiler, and thus the overall cost of the run-time check should be insignificant.

- ☐ A run-time check at the end of a value-returning function ensures that it exits with an explicit `return` or `throw`.

It is illegal in C++ for a value-returning function to exit without an explicit `return` or `throw`, but implementations are not required to check for that. In most implementations, a function exiting without an explicit `return` or `throw` will return some undefined value to the caller, typically whatever happens to be in the return-value register. A pointer-valued function could thus return an illegal value.

The check is trivial to implement—after the function's last statement, the compiler generates an unconditional error call. The check imposes no run-time penalty, and in most situations, dead-code removal will eliminate the check entirely. Good compilers should warn the programmer about functions that may exit without a `return` or `throw` (that is, those functions in which dead-code removal isn't able to eliminate the check).

- ☐ All pointers must be explicitly initialized. In particular:

  All pointer-valued variables and pointers created by `new` must be explicitly initialized.

  All pointer-valued non-static members of an object must be explicitly initialized by an aggregate list, a member initializer of a constructor, or a compiler-synthesized copy constructor. A constructor's member initializer for a pointer- or reference-containing member must not refer to `this` or other members of its class or base classes. A compiler-synthesized implicit default constructor does not explicitly initialize pointer-valued members.

  As a special case, arrays of pointers are automatically initialized to null.

Pointers must be explicitly initialized to legal values, since C++ does not define the initial values of automatic variables or members of objects created by `new`. C++ already requires references to be initialized.

In practice, requiring explicit initializers won't produce less efficient code, since good compilers remove clearly useless assignments. But some programmers will grumble about having to write a few extra characters in constructors and variable declarations.

The language would be more concise if implementations automatically initialized all pointers to null, optimizing away any useless assignments. Unfortunately, many programmers would object if this change were made to both the safe and the unsafe parts of the language, since compilers can't always detect when an initializing assignment can be optimized away, and the programmers would object to the (very small) inefficiency. But if pointers were initialized only in the safe subset, code that worked in the safe subset might behave differently if copied to unsafe code. For example:

```
char *s;
if (s == 0) s1 else s2;
```

The statement `s1` would be executed in the safe subset, while `s1` or `s2` may be executed in the unsafe language. While such behavior is consistent with the definition of C++, as a practical matter it seems unwise to create situations that allow safe code free of checked run-time errors to behave differently when copied to an unsafe scope.

Unfortunately, C++ provides no general method for initializing arrays of pointers. Thus, arrays of pointers declared in the safe subset will be automatically initialized to null by the compiler (see section 7.7).

Some people have suggested that `new` should zero all objects, not just arrays of pointers. But it's more efficient to require explicit initialization, since a class often needs to initialize a member to a non-zero value. Zeroing every object could slow down `new` (or garbage collection) quite a bit. For example, a good implementation of `malloc`/`free` (which doesn't zero objects) may take about 70 RISC instructions, while zeroing a 200-byte object could add at least 50 more instructions.

Given a pointer type `T`, the standard syntax `new T(e)` initializes the newly allocated pointer.

A constructor's member initializer for a pointer-containing member mustn't refer to other members of its class, because those members may not be initialized yet. Consider:

```
struct T {
    int* p1;
    int* p2;
    T(): p1(p2), p2(p1) {}; /* error in the safe subset */
    };
```

The member `p1` gets initialized to the uninitialized value of `p2`, and then `p2` gets initialized to the still uninitialized `p1`. It isn't feasible for compilers to detect all such situations; for example:

```
struct A {
    virtual char* f() {return 0;}
    char* g();};

char* A::g() {return f();};

struct B: A {
    char* p;
    virtual char* f() {return p;}
    B(): p(g()) {}};
```

`B`'s member initializer for `p` calls `A::g()`, which calls `f()`, which is bound to `B::f()`, which returns the uninitialized `p`.

A member initializer mustn't refer to `this`, since it could be passed to a function that returns an uninitialized member.

7.4. Restricting built-in arrays

- ☐ Given a pointer expression `p`, expressions of the form `p + c`, `p - c`, `p++`, `p--`, `++p`, and `--p` are disallowed.

We know of no efficient way for ensuring that `p + c` results in a legal pointer within the bounds of its referent array. Most schemes involve carrying some information along with the pointer, doubling the natural size of the pointer and imposing a cost on every dereference [Steffen 92, Dix 93].

Safe arrays provide an adequately expressive replacement for pointer arithmetic. With modern optimizing compilers, pointer arithmetic is no longer necessary to produce efficient code. As Stroustrup points out in *The C++ Programming Language* [Stroustrup 91, page 93], the only remaining purpose for pointer arithmetic is syntactic conciseness—programmers like writing tight little custom loops for scanning strings and copying arrays. But with the introduction of powerful string and array classes and the use of function inlining, programmers can get more conciseness (and often more efficiency) by using the searching and copying operations provided by those abstractions.

More than a few C++ enthusiasts will balk at giving up `p++`, but trading a small amount of conciseness for safety is well worth it, at least to any programmer who has spent days tracking down a single storage bug.

- ☐ Expressions of the form `a[i]` are disallowed.

Subscripting is equivalent to pointer arithmetic—`a[i]` means the same as `*(a + i)`. The standard array classes provide safe array subscripting (section 7.8).

- ☐ An expression of type "array of `T`" (`T[] or T[c]`) cannot be converted to type `T*`.

Converting an automatic array to a pointer to its first element could create a dangling pointer after the function defining the array exits. The operator `&` can be used to take the address of non-automatic arrays.

- ☐ A formal parameter of type `T[]`, `T[c]`, or `T(&)[]` cannot be initialized with an argument, unless the argument is a string literal. A formal parameter of type `T(&)[c]` can be initialized only with an argument of type `T[c]` or `T(&)[c]`.

In general, there is no safe use of a formal parameter of type "array of `T`", since the length of the actual argument isn't supplied and needn't match the size of the formal. An argument can be passed safely to a formal of type `T(&)[c]` (a reference to an array with known size), since the language requires the argument to be an array of the same size. (This contrasts with a formal of type `T[c]`, which can be initialized with an array argument of any size.)

The one exception for string literals allows the standard string class `Text` to provide a safe constructor `Text(const char s[])` that converts string literals to `Texts`. The constructor's interface is safe, meaning its use by clients can't violate safety, but it must be implemented using unsafe language features (the constructor uses the unsafe function `strlen` to find the string's length).

7.5. Preventing fabrication of invalid pointers

- ☐ All casts to types containing pointers, references, or functions are disallowed, except for widening casts and checked narrowing casts. Given a class `B` derived from class `A`, a cast is widening if it casts from `B*` to `A*`, `B*` to `B*`, or `B*` to `void*`; a cast is narrowing if it casts

from `A*` to `B*`. As in general with C++, two types are considered the same here if they differ only in instances of `const`, `volatile`, or `gc`. To be considered a widening or narrowing cast, the class types involved must be fully declared at the point of the cast.

We assume the ANSI C++ committee will soon adopt a proposal for checked narrowing casts [Stroustrup 92], which can't violate safety.

Casts that fabricate pointers or references to non-existent objects must be disallowed. Casting to a function type can create a function that does an implicit disallowed cast; for example, on machines where pointers and integers have the same size, casting a function of type `void (int*)` to `void (int)` creates a function that implicitly casts its argument from `int` to `int*`.

☐ The declaration of union types containing pointers, references, or functions is disallowed.

C++ unions are untagged, leaving no efficient way for implementations to effectively check which variant of a union variable is currently being used. Without run-time checking the type `union {S s; T t;}` could be used to arbitrarily cast from a non-pointer type `T` to a pointer type `S`. As discussed in appendix B, it isn't feasible to have compilers add implicit tags to unions.

Class derivation provides a better alternative to many uses of unions, without sacrificing either time or space efficiency. Untagged pointer-containing unions remain most useful for low-level systems code and applications that must deal with predefined structure formats; even here, the use of such unions can usually be hidden behind a safe interface (whose implementation is declared unsafe) [Stroustrup 91, page 169].

☐ Overloading `operator new`, either globally or for particular classes, is disallowed.

When an overridden `operator new` is invoked, the language does an implicit cast from the `void*` return type of `operator new` to the argument type of the call to `new`. Nothing would prevent a buggy `operator new` from implicitly casting the result from one pointer type to another, violating safety. We know no practical method for ensuring the safety of that cast.

Safe code has no need for overriding `operator new`, since the safe subset must necessarily use garbage collection to enforce safety. The language interface (section 4.1) prevents `gc` classes from overriding a class's `operator new`.

☐ Functions declared with ellipsis (`f(...)`) are disallowed.

Functions with varying numbers of arguments are unsafe, since the function relies on the caller to follow whatever convention is established to signal the number of arguments; a buggy caller could cause unpredictable results in the function as it tried to access its arguments. And using the `va_arg` macro would, in general, require unsafe casting.

The most common use of variadic functions is to call C's `printf` and friends. But the standard C++ stream library provides an adequately concise and efficient replacement. Further, `printf`-style formatting strings could be added easily to C++ streams; for example:

```
cout("i = %3d  x = %7.3f\n") << i << x;
```

Default arguments and function overloading subsume most other uses of variadic functions in C.

7.6. Preventing dangling pointers and references

☐ The operator `delete` and explicit calls to destructors are disallowed.

Obviously, `delete` can cause dangling pointers and references, and we know of no way to check for dangling pointers that doesn't involve increasing the size of pointers and checking every pointer dereference at run-time. In effect, garbage collection calls `delete` automatically when an object is no longer accessible.

Informally, destructors turn objects into “raw memory”. Accessing a destroyed object is explicitly undefined by the language, and there is no way to ensure such accesses don't violate GC-safety without referring to particular implementations.

- Static and run-time checks ensure that in the expression `&e`, `e` is not the lvalue of an automatic variable.

Taking the address of an automatic variable would leave the resulting pointer dangling after the variable's function exits.

The compiler can detect statically most improper uses of `&e`. Specifically, `e` cannot be the lvalue of an automatic variable if:

> `e` is a static variable, possibly followed by one or more applications of the operator “`.`”;
> or
> `e` is formed by one or more applications of the operators `*` and `->`; or
> `e` is a reference initialized to an lvalue known not to be an automatic variable.

The second rule follows from the invariant enforced by the safe subset: pointers never reference automatic variables. In all other cases, the compiler must assume `e` could be an automatic variable's lvalue and generate a run-time check. For example, if `e` is a reference-valued parameter or function-call result, it might well reference an automatic variable.

Checking if a pointer addresses an automatic variable is equivalent to checking if it points into a thread stack, and that check is fairly cheap. In most Unix implementations, the stacks are allocated from a dedicated virtual-memory segment with statically known bounds. Testing the address against those bounds involves two comparisons, which would take four cycles on the MIPS R3000 architecture.

Even in implementations where stacks are intermingled with the heaps, the check can be implemented cheaply. To support conservative scanning of the stacks and registers, collectors typically use a “page table” of some sort that tracks which pages are allocated to the heap. This same table could also record which pages are allocated to stacks. The run-time check can extract the page index from the address and look it up in the table. This test would take about 7 to 10 cycles on the MIPS architecture, depending on whether the operating system allows the data segment to be very sparse. (Sparse data segments require a two-level table.)

But note that in practice, the great majority of uses of the `&` operator will not require a run-time check, so the actual overhead of the run-time check will be quite small.

- Static and run-time checks ensure that any references in an object created by `new` are not initialized to an automatic variable.

Storing a reference to an automatic variable in a heap object would create a dangling reference after the variable's function exits. Using the same static and run-time checks as for the `&` operator, the compiler must ensure the initializing expressions of all references in the newly created object don't refer to an automatic variable. In practice, heap objects containing references are not common, and of those, most are initialized with expressions that the compiler can statically determine are not automatic lvalues. Thus, the overhead of the checks will be quite small.

- If `this` is used explicitly and is not an operand of `->` or `*`, a run-time check ensures it does not point at an automatic variable.

When a member function of an automatic object is called, `this` gets bound to the address of the variable on the stack. Storing `this` in another variable, returning it, or passing it to another function might then create a dangling pointer. The run-time check ensures `this` cannot escape outside of its scope. Implicit use of `this` to access members and use of `this` as an operand of `*` or `->` are not restricted and require no run-time check.

The restriction on `this` will not significantly affect the expressiveness of C++. The referent of `this` can be passed efficiently to a function `f` by reference:

```
void f(T& t);

class T {
    m() {...f(*this)...}; ...};
```

Note that this use requires no run-time check.

Storing `this` in a static or heap object is a useful technique for managing lists of active objects, and the safe subset requires such objects to be allocated in the heap. Programmers might object to this requirement, thinking that their particular objects might be more efficiently allocated on the stack. But in practice, when a class maintains active lists of its objects, the objects tend to be long-lived, and the performance penalty for allocating them in the heap is very small relative to total execution time.

The run-time check of `this` is the same as the check used for `&` and `new`. We can make a crude estimate of the overall cost of the check of `this`. A good RISC calling sequence consumes about 10% of total execution time, and on the MIPS, the average call takes about 8 instructions, about twice the cost of the run-time check of `this`. If 1/10 of all calls make explicit references to `this` requiring checks, then the checks will increase total execution time by about 0.5%. In practice, far fewer than 1/10 of all function calls in an average program will make checked references to `this`.

7.7. Compiler restrictions

- ☐ If a temporary object is created anywhere within the initializing expression of a reference, the temporary's storage must not be reused until the block containing the initialization is exited.

This restriction on the compiler prevents dangling references to temporary objects. Consider this example:

```
struct T {...};
T& f(T& t) {return t;}
T& t = f(T());
```

A temporary object is created as the argument to `f`. If the compiler destroys the temporary before the block containing it exits, `t` would be left dangling. But as long as `T`'s destructor is written in the safe subset and the compiler doesn't reuse the temporary's storage until the block exits, then the temporary object will continue to contain valid pointers and it's use through `t` couldn't violate safety.

This restriction may already be required by the C++ definition. *The Annotated C++ Reference Manual* says only that “If a reference is bound to a temporary, the temporary must not be destroyed until the reference is” (12.2). Do the authors intend this to apply to examples like that above, where the compiler doesn't know whether the result of an initializing function call is a temporary or not?

The ANSI committee is trying to define exactly when temporaries should be destroyed, but no matter what they decide and no matter what the current interpretation of the reference manual is, this implementation restriction will be compatible and will ensure that GC-safety is preserved. It's trivial for compilers to implement—at least some already behave this way.

- ☐ Arrays of pointers must be initialized to null.

C++ provides no general method to initialize arrays of pointers that can be easily enforced by the safe subset. Consider a class containing an array of pointers—the lack of general array-valued expressions means that the array cannot be initialized by a constructor's member initializer. Since there is no easy way for the compiler to ensure that the programmer has initialized an array, the