Proposal to Acknowledge that Garbage Collection for C++ is Possible

X3J16/96-0114
WG21/N0932

Bjarne Stroustrup

AT&T Research

Murray Hill

New Jersey 07974

USA

The ARM, "The C++ Programming Language (2nd edition)", and "The Design and Evolution of C++" all mention that automatic garbage collection of C++ is possible, but that an implementation is not required to provide a garbage collector. This is a proposal to make this explicit in the standard and to specify a couple of details of what it means to collect garbage. The proposal is for clarification rather than significant normative changes.

Introduction

The most loudly proclaimed fundamental weakness of C++ these days is the absence of automatic garbage collection. This criticism is harming C++’s reputation, scaring potential users away, and restricting the range of applications for which C++ is a strong contender as an implementation language. For many applications automatic garbage collection is indeed a very powerful tool and the right tool for the job.

It is my firm opinion that the complaints about the lack of garbage collection in C++ will continue, will become louder as more alternative languages provide garbage collection, and that the cases where the complaint is based on valid reasons will become more frequent.

My defense has traditionally been that garbage collection is possible for C++ and that garbage collecting implementations do indeed exist. Currently both public domain and commercial garbage collectors for C++ are shipping. Nothing in the definition of C++ requires garbage collection - and if it did serious harm would be done to many C++ users - but on the other hand nothing prevents it from being done either. See D&E for a fairly detailed discussion of C++ and garbage collection.

However, the "permitted but not required" position is weakened by being implicit rather than explicit in the draft standard and by a couple of possible alternatives in the meaning of "garbage collection for C++." I therefore propose to make "permitted but not required" explicit by defining what it means to be "garbage."

My proposal reflects my opinion that we now have seen enough implementations to know that automatic garbage collection is viable for C++ and also a convergence of opinion of how to handle the unspecified cases. I expect to see a significant increase in the use of garbage collectors with C++ over the next few years. Providers and users of garbage collectors or C++ would like their implementations to be standards conforming and to know precisely what that means.

Most of the suggestions below are clarifications and only one affects conformance.

Garbage

The fundamental idea of garbage collection is that an object that is no longer referred to in a program will not be accessed again and can therefore be destroyed and its memory reused for some new object.

For example:

```
int p = new int;
p = new int;
```

Here the first int created is unreferenced and its memory can be used for some other new object.

What should it mean for an object to be unreferenced? Consider:

```
int* p = new int;
long p1 = reinterpret_cast<long>(*p)&0xFFFFOOOO;
long p2 = reinterpret_cast<long>(*p)&0xOOOOFFFF;
p = 0;
// #1:  no pointer to the int exist here

p = reinterpret_cast<int*>(p1|p2);
// now the int is referenced again.
```

Often, pointers stored as non-pointers in a program are called "disguised pointers." In particular, the pointer originally held in "p" is disguised in p1 and p2 above. Is a garbage collector allowed to deallocate the int formerly pointed to by p at #1 above?

I propose that such program should be considered implementation dependent. In particular, I believe that C and C++ currently consider such programs well formed (though I would be delighted if I was wrong), but that a garbage collecting implementation should be allowed to collect objects to which no object of pointer or reference type refers. That is, a disguised pointer is not sufficient to keep an object alive.

I believe that the example is already implementation dependent because there is no guaranteed conversion from pointer to int.

Unions

A special and difficult issue is how to handle a union with a pointer and a non-pointer member. Consider:

```
union U {
        char* p;
        int i;
};

U u;

u.p = new char;
// #1
delete u.p;
u.i = reinterpret_cast<int>(new char);
// #2
char* p = reinterpret_cast<char*>(u.i);
```

Should a pointer be considered disguised if it is placed in a union

that also has an integer member? Should any integer in a union that has a pointer member be considered a pointer?

If a pointer to an object is placed in the pointer member, that object must still be considered referenced. Thus collecting the char at point #1 would be an error.

Once the only pointer to 'new char' was cast to an int it was disguised and that int object is unreferenced. Placing the cast value in a union is a red herring. Thus collecting the int at point #2 is ok.

However, it is usually impossible for a garbage collector to distinguish these two cases so a collector will usually have to be conservative and assume that any value of a union a potential pointer to an object on the heap. This is an implementation issue that doesn't concern the standard.

Pointer to "the Middle of" Objects

Some garbage collectors consider an object to be unreferenced unless a pointer to the first byte of the object exists. This is unacceptable for C++. Consider:

```
class A { int a; };
class B { int b; };
class C : public A, public B { int c; };

B* p = new C;
```

Here, p most likely points to "the middle" of the C object and no pointer to the first byte of the C object exists of need to exist.

A pointer to any part of an object is sufficient for that object to be considered referenced.

Similarly, any valid pointer to an array (even a pointer to one past the end of the array) is considerd a valide pointer to the array and the complete array is considered referenced.

Delete

If an implementation automatically collects garbage, the 'delete' and 'delete[]' operators are no longer needed to free memory for potential reuse. Thus, a user relying on garbage collection could simply refrain from using these operators. However, in addition to freeing memory 'delete' and 'delete[]' invoke destructors.

In the presence of a garbage collector, 'delete p' invokes the destructor (if any) as ever, but any reuse of the memory is postponed until it is collected. In many cases, this prevents serious errors caused by multiple deletes of the same object. Specifically, destructors that can be repeatedly executed become harmless.

As ever, access to an object after it has been deleted is undefined. In particular, operator delete() may write to the memory previously occupied by the object. I do not propose to change this, but observe that by delaying reuse of storage until collection time, access to a deleted object (through another pointer to the object) that has no destructor or a destructor that performed deletes only is far less likely to lead to disasterous results.

Destructors

When an object is about to be recycled by a garbage collector two alternatives exists:

(1) call the destructor (if any) for the object.
(2) treat the object as raw memory (don't call its destructor).

In the ARM, I proposed (2) because one can see a garbage collector as a mechanism for simulating an infinite memory and undeleted objects never gets deleted. I also proposed that objects for which the destructor should be called should be specifically registered. Experience with garbage collectors seems to have proven that not calling destructors at collection time is the right default, but that there is a need for some destructors to be called at collection time. However, there is no consensus on how to express a need for a destructor to be called.

Consequently, I propose (2), leave the mechanism for requesting constructors to be called by the collector (''finalizable objects'') unspecified, and to introduce a macro to help programmers isolate declarations and/or code relating to finalization actions beyond the scope of the standard.

__COLLECTING

It should be implementation defined whether an implementation provides a garbage collector or not. Should the programmer be able to tell whether an implementation collectes or not without doing experiments with disguised pointers or timers?

I can imagine programmers wanting to know and even to write code that depends on whether a garbage collector is available or not. In fact, I am confident that programmers will write code to try to determine whether an implementation is garbage collecting or not whatever the standard says so that the absence of a standard mechanism will simply lead to incompatibilities.

Consequently, I propose a macro __COLLECTING with the value 0 if no collector is used and 1 if a collector as described here is used. Different values are left undefined with the expectation that they will be used by more radical experiments with garbage collection.

For example:

```
#if __COLLECTING==0
        // error: garbage collector assumed by this program
#endif
```

or

```
class X {
        int* p;
        // ...
#if __COLLECTING==0
        ~X() { delete p; }
#endif
        // ...
};
```

I dislike macros, but I suspect adding one to ease the use of garbage collection for C++ is worth while.

Working Paper Text

The only part of this proposal that actually affects conformance is

the presence of the __COLLECTING macro, and the exact detailed meaning of that cannot be specified. The rest of the proposal above are clarifications that might be scattered in the text. However, since the aim of this proposal is to make to possibility of garbage collection explicit, I propose to define what it means for an object to be reachable.

Add:

An object is referenced if it is named or if a pointer or a reference type holds the address of any part of that object. A pointer stored in a union through a pointer member of that union is considered a pointer. A value of a non-pointer type obtained by casting a pointer is not considered a pointer.

comment:
	For example:

```
union U {
	char* p;
	int i;
};

U u;

u.p = new char;
// the char pointed to by u.p is referenced

u.i = reinterpret_cast<int>(new char);
// the char allocated on the previous line is unreferenced
// the char with the address originally assigned to u.p
// is unreferenced
```

end_comment

An object is reachable if
	(1) it is a named object
	(2) it is referenced by a reachable object.

comment:
	For example:

```
int* p = new int;
long p1 = reinterpret_cast<long>(*p)&0xFFFFOOOO;
long p2 = reinterpret_cast<long>(*p)&0xOOOOFFFF;
p = 0;
// now the integer originally referred
// to by p is unreferenced and unreacheable.

p = reinterpret_cast<int*>(p1|p2);
// now the int is referenced again.
```

end_comment

An implementation may use the memory for an unreachable object for another object. An implementation that does that is called garbage collecting.

Access to an object that has been unreachable at any point is implementation defined.

It is implementation defined whether an implementation is garbage collecting or not. If an implementation is garbage collecting, the macro __COLLECTING is defined as 1 in header <new>; otherwise __COLLECTING is 0.