```
    tag = v.tag;
    u = v.u;
    return *this; }

template< class T0, class T1 >
inline Variant2< T0, T1>&
Variant2< T0, T1 >::operator =( const T0& t0 ) {
    tag = 0;
    this->u.t0 = t0;
    return *this; }

template< class T0, class T1 >
inline Variant2< T0, T1 >&
Variant2< T0, T1 >::operator =( const T1& t1 ) {
    tag = 1;
    this->u.t1 = t1;
    return *this; }

template< class T0, class T1 >
inline void Variant2< T0, T1 >::Error( const char message[] ) {
    cerr << message << "\n";
    abort(); }

#endif /* _Variant_h */
```

Appendix F: WeakPointer.h

This appendix presents the standard interface `WeakPointer.h`, which provides weak pointers and object clean-up.

```
#ifndef   _WeakPointer_h_
#define   _WeakPointer_h_

/**********************************************************************
```

WeakPointer and CleanUp

```
**********************************************************************/

#pragma safe

/**********************************************************************
```

WeakPointer

A *weak pointer* is a pointer to a heap-allocated object that doesn't prevent the object from being garbage collected. Weak pointers can be used to track which objects haven't yet been reclaimed by the collector.

```
**********************************************************************/


template< class T > class WeakPointer {
public:

WeakPointer( T* t = 0 );
      /* Constructs a weak pointer for *t, reactivating it if it was previously deactivated (see
      below). t may be null. It is an error if t is non-null and *t is not a collected object. */

T* Pointer();
      /* wp.Pointer() returns the original pointer from which wp was constructed or null
      if wp has been deactivated. The collector deactivates a weak pointer when it detects that
      the referenced object is unreachable by normal pointers. (Reachability and deactivation are
      defined below.) */

int operator==( WeakPointer< T > );
      /* Given T* t1 and T* t2, t1 == t2 if and only if WeakPointer<T>(t1) ==
      WeakPointer<T>(t2). */

int Hash();
      /* Returns a hash code suitable for use by multiplicative- and division-based hash tables.
      If wp1 == wp2, then wp1.Hash() == wp2.Hash(). */
};

/**********************************************************************
```

CleanUp

A garbage-collected object can have an associated clean-up function that will be invoked some time after the collector discovers the object is unreachable via normal pointers. Clean-up functions can be used to release resources such as open-file handles or window handles when their containing objects become unreachable. The initial clean-up function of a collected object is its destructor.

There is no guarantee that the collector will detect every unreachable object (though it will find almost all of them). Clients should not rely on clean-up to cause some action to occur—clean-up is only a mechanism for improving resource usage.

Every object with a clean-up function also has a clean-up queue. When the collector finds the object is unreachable, it enqueues it on its queue. The clean-up function is applied when the object is removed from the queue. By default, objects are enqueued on the garbage collector's queue, and the collector removes all objects from its queue after each collection. If a client supplies another queue for objects, it is his responsibility to remove objects (and cause their functions to be called) by polling it periodically.

Clean-up queues allow clean-up functions accessing global data to synchronize with the main program. Garbage collection can occur at any time, and clean-ups invoked by the collector might access data in an inconsistent state. A client can control this by defining an explicit queue for objects and polling it at safe points.

The following definitions are used by the specification below:

> Given a pointer `t` to a collected object, the *base object* `BO(t)` is the value returned by `new` when it created the object. (Because of multiple inheritance, `t` and `BO(t)` may not be the same address.)
>
> A weak pointer `wp` references an object `*t` if `BO(wp.Pointer()) == BO(t)`.

```
**********************************************************************/

template< class T, class Data > class CleanUp {
public:

static void Set( T* t, void c( Data* d, T* t ), Data* d = 0 );
      /* Sets the clean-up function of object BO(t) to be <c, d>, replacing any previously
      defined clean-up function for BO(t); c and d can be null, but t cannot. Sets the clean-up
      queue for BO(t) to be the collector's queue. When t is removed from its clean-up queue,
      its clean-up will be applied by calling c(d, t). It is an error if *t is not a collected
      object. */

static void Call( T* t );
      /* Sets the new clean-up function for BO(t) to be null and, if the old one is non-null,
      calls it immediately, even if BO(t) is still reachable. */

class Queue {public:
    Queue();
       /* Constructs a new queue. */

    void Set( T* t );
       /* q.Set(t) sets the clean-up queue of BO(t) to be q. */

    int Call();
       /* If q is non-empty, q.Call() removes the first object and calls its clean-up
       function; does nothing if q is empty. Returns true if there are more objects in the queue.
       */
    };
};

/**********************************************************************
```

Reachability and Clean-up

An object is *reachable* if it can be reached via a path of normal pointers starting at the registers, stacks, global variables, or some other object `O` with a non-null clean-up function (even if `O` is not reachable).

This definition of reachability ensures that if object `B` is accessible from object `A` (and not vice versa) and if both `A` and `B` have clean-up functions, then `A` will always be cleaned up before `B`. Note that as long as two objects reference each other directly or indirectly via normal pointers and both have non-null clean-up functions, then the objects will be reachable and they won't be collected.

When the collector finds an unreachable object with a null clean-up function, it atomically deactivates all weak pointers referencing the object and recycles its storage. If object `B` is accessible from object `A` via a path of normal pointers, `A` will be discovered unreachable no later than `B`, and a weak pointer to `A` will be deactivated no later than a weak pointer to `B`.

When the collector finds an unreachable object with a non-null clean-up function, the collector atomically deactivates all weak pointers referencing the object, redefines its clean-up function to be null, and enqueues it on its clean-up queue. The object then becomes reachable again and remains reachable at least until its clean-up function executes.

The clean-up function is assured that its argument is the only accessible pointer to the object. Nothing prevents the function from redefining the object's clean-up function or making the object reachable again (for example, by storing the pointer in a global variable).

If the clean-up function does not make its object reachable again and does not redefine its clean-up function, then the object will be collected by a subsequent collection (because the object remains unreachable and now has a null clean-up function). If the clean-up function does make its object `*t` reachable again, any subsequent calls to `WeakPointer<T>(t)` will reactivate the original weak pointer and return it. Further, if a clean-up function is redefined for `t`, then it will be invoked on the object the next time the collector finds it unreachable.

Note that a destructor for a collected object cannot safely redefine a clean-up function for its object, since after the destructor executes, the object has been destroyed into “raw memory”. (In most implementations, destroying an object mutates its vtbl.)

An implementation sketch

Here's a sketch of an implementation suitable for incremental and concurrent collectors:

The collector maintains a single table `wps`, representing both weak pointers and registered clean-up functions, for all instances of the `CleanUp` template class:

```
struct Entry {
    void*           object;
    int             timeStamp;
    CleanUpFunction cleanUp;
    void*           data;
    CleanUpQueue    q;
    int             next; };
Entry wps[];
```

A weak pointer is represented as two integers:

```
template<class T> class WeakPointer {
    ...
    int index, timeStamp;};
```

The entry `e = wps[wp.index]` describes weak pointer `wp`: `wp` is activated only if `wp.timeStamp = e.timeStamp` (otherwise, the entry has been reused for another weak pointer and `wp` is deactivated). `e.object` is the referent of `wp`. The implementation of `wp.Pointer()` looks like:

```
template<class T> WeakPointer::Pointer() {
    if (timeStamp == wpTable[index].timeStamp)
        return (T*) wpTable[index].object;
    else
        return 0;}
```

`CleanUp<T, D>::Set(t, c, d)` implicitly creates a weak pointer to `BO(t)` and sets its entry's `cleanUp`, `data`, and `q`.

There is at most one `WeakPointer<T>` referencing a given object `T* t`, but many weak pointers of different types may address different sub-objects of the same base object `BO(t)`. All the entries `e` in `wps` referring to the same base object `bo` (`BO(e.object) == bo`) are chained through their `next` field. There will be at most one entry in a chain with a non-null `e.cleanUp`, and if it there is one, it will be at the head.

A hash table `boToIndex` maps a base-object address to the index of the head of the chain of `wps` entries for that base-object address:

```
Table<void*, int> boToIndex;
```

Unused entries in `wps` are chained through their `next` fields.

For a base object `bo` with a null clean-up, the collector deactivates its weak pointers by deleting `bo` from `boToIndex` and, for every entry `e` in `bo`'s `wps` chain, setting `e.object` to null and adding `e` to the list of unused entries. If `bo` has a non-null clean-up, the collector deactivates its weak pointers by setting `e.timeStamp = -e.timeStamp` for every entry `e` in `bo`'s chain, . The weak pointers may be later reactivated, so the entries can't be deleted yet.

Here's an outline of how a mark-and-sweep collector identifies objects needing clean-up and deactivates weak pointers. Similar methods can be used with copying collectors:

1. The collector performs the normal mark phase of its collection, marking all objects reachable from the roots. The table `wps` is not treated as a root.

2. At the end of normal marking, the collector passes over `wps`, recursively marking all objects that are reachable from an object with a non-null clean-up. An object with a clean-up is marked only if it is reachable from some other object with a clean-up. (This marking requires a careful linear-time implementation to meet the specification of “reachability” given above. In particular, extra mark bits per object appear necessary.)

3. For each chain of entries referenced by a `<bo, index>` pair in `boToIndex`, if the chain references an unmarked object with a null clean-up, the collector deactivates the weak pointers in the chain. If the chain references an unmarked object with a non-null clean-up, the collector deactivates the weak pointers, marks the object, sets its clean-up to be null, and enqueues the object on its clean-up queue.

4. After collection finishes, the collector removes all entries from the default clean-up queue via `while (q.Call())`.

The compiler generates a clean-up function for each `gc class T` that has an explicit destructor or that has a base class with an explicit destructor:

```
class T ...{
    static void __CleanUp(void* data, void* t) {
        ((T*) t)->~T();}};
```

For an expression `new T`, the compiler generates a call to register the clean-up function with the newly created base object. For an expression `new T[e]`, the compiler generates a clean-up function that invokes the destructors of each element of the array.

```
************************************************************************/


#ifdef _IGNORE_
/************************************************************************
```

The C interface

This version has the same semantics as the C++ version, except that it cannot use templates for safe typechecking. The C++ version probably is implemented using this interface.

```
************************************************************************/

typedef struct {int index, timeStamp;} WeakPointer;
      /* The struct fields are private to WeakPointer. */

WeakPointer WeakPointer_New( void* t );
void* WeakPointer_Pointer( WeakPointer wp );

typedef void CleanUp_Function( void* data, void* t );
void CleanUp_Set( void* t, CleanUp_Function cleanUp, void* data );
void CleanUp_Call( void* t );

typedef void* CleanUp_Queue;
CleanUp_Queue CleanUp_Queue_New();
void CleanUp_Queue_Set( CleanUp_Queue q, void* t );
void CleanUp_Queue_Call( CleanUp_Queue q );

#endif /* _IGNORE_ */

#endif /* _WeakPointer_h_ */
```