```
int a[] = {39, 45, 57, 63, 79};
SubArray<int> s = SUB(int, a);
```

The form `SUB2(type, array, d2)` creates a `SubArray<Array<type, d2> >` that accesses the elements of a two-dimensional array whose second dimension has `d2` elements. For example,

```
int a[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};
SubArray<Array<int, 4> > s = SUB2(int, a, 4);
```

creates a `SubArray s` that accesses the 2 x 4 array `a`. The forms `SUB3(type, array, d2, d3)` and `SUB4(type, array, d2, d3, d4)` operate on three- and four-dimensional arrays.

Implementation notes

These template classes are implemented without using class derivation or abstract base classes. The benefits of derivation would accrue only with the use of virtual functions, but virtual functions would add a word of overhead to every array. This is intolerable, since `Arrays`, `SubArrays`, and `DynArrays` are intended to be as efficient as built-in C++ arrays. The approach used here, three separate classes and implicit conversions between them, is just as expressive for the programmer but more efficient.

With a good compiler, the only overhead of using these classes instead of the built-in C++ equivalents comes from the run-time checks.

Run-time checking

The run-time checks can be disabled at compile time by defining the preprocessor variable `CHECKING=0` before including this file. Of course, disabling checking eliminates the guarantee of safety provided by the safe subset. In general, programmers should leave the checks enabled as late in the development process as possible, even perhaps in production code. With run-time checking, a buggy program halts as soon as safety is violated, before it can produce incorrect results.

The exact mechanism of signaling run-time errors is not specified by this interface. In general, a run-time error (such as an out-of-bounds index) is a bug, and programs should not try to handle such errors.

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

#pragma safe

#include <stdlib.h>

template< class T > class SubArray {
public:
    int Number();
    T& operator [](  int i );
    SubArray< T > Sub( int start, int number );
    SubArray< T > Sub( int start );
    SubArray< T > Sub();
    void Copy( SubArray< T > );
    int operator ==( SubArray< T > );
    int operator !=( SubArray< T > );
    int Equal( SubArray< T > );

#pragma unsafe
    SubArray( T* t, int number );
       /* Creates a SubArray of number elements at t, t + 1, ..., t + number - 1. */
```

```
/* private: These members should be private, but C++ doesn't allow us to say friend class
Array< T, n > for all n;. But they are declared unsafe and can't be accessed in safe code. */

    T (&t)[];                       /* The array elements */
    int number;                     /* Number of elements */
    T* First();                     /* Returns first element of array */
    void Error( const char [] );    /* Signals a run-time error */
    };


template< class T, int n > class Array {
public:
    Array();
    Array( SubArray< T > );
    int Number();
    T& operator []( int i );
    SubArray< T > Sub( int start, int number );
    SubArray< T > Sub( int start );
    SubArray< T > Sub();
    operator SubArray< T >();
    void Copy( SubArray< T > );
    int operator ==( SubArray< T > );
    int operator !=( SubArray< T > );
    int Equal( SubArray< T > );

private:
       T  a[ n ];                    /* The elements */
    T* First();                   /* Returns the first element */
    void Error( const char [] ); /* Signals a checked run-time error */
    };


template< class T > class DynArray {
public:
    DynArray();
    DynArray( int number );
    DynArray( SubArray< T > );
    int IsNull();
    int Number();
    T& operator []( int i );
    SubArray< T > Sub( int start, int number );
    SubArray< T > Sub( int start );
    SubArray< T > Sub();
    operator SubArray< T >();
    DynArray< T > operator =( DynArray< T > );
    void Copy( SubArray< T > );
    int operator ==( DynArray< T > );
    int operator ==( SubArray< T > );
    int operator !=( DynArray< T > );
    int operator !=( SubArray< T > );
    int Equal( SubArray< T > );

private:
    T* t;                         /* First element of array */
    T* First();                   /* Returns the first element */
    void Error( const char [] ); /* Signals a checked run-time error */
    typedef gc T GCT;
    };

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

Inline implementation

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

```
#pragma unsafe

#ifndef CHECKING
#define CHECKING 1
#endif

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

SubArray

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

template< class T >
inline int SubArray< T >::Number() {
    return number; }

template< class T >
inline T& SubArray< T >::operator []( int i ) {
    if (CHECKING && (i < 0 || i >= Number())) Error( "[]" );
    return t[ i ]; }

template< class T >
inline SubArray< T > SubArray< T >::Sub( int start, int number ) {
    if (CHECKING && (start < 0 || start + number > Number()))
        Error( "Sub" );
    return SubArray< T >( First() + start, number ); }

template< class T >
inline SubArray< T > SubArray< T >::Sub( int start ) {
    return Sub( start, Number() - start ); }

template< class T >
inline SubArray< T > SubArray< T >::Sub() {
    return Sub( 0, Number() ); }

template< class T >
inline void SubArray< T >::Copy( SubArray< T > s ) {
    int n = Number();
    if (CHECKING && n != s.Number()) Error( "Copy" );
    T* to   = First();
    T* from = s.First();
    for (int i = 0; i < n; i++) *to++ = *from++; }

template< class T >
inline int SubArray< T >::operator ==( SubArray< T > s ) {
    return Number() == s.Number() && First() == s.First(); }

template< class T >
inline int SubArray< T >::operator !=( SubArray< T > s ) {
    return ! (*this == s); }

template< class T >
inline int SubArray< T >::Equal( SubArray< T > s ) {
    int n = Number();
    if (n != s.Number()) return 0;
    T* to   = First();
    T* from = s.First();
    for (int i = 0; i < n; i++) if (*to++ != *from++) return 0;
    return 1; }

template< class T >
inline SubArray< T >::SubArray( T* first, int n )
    : t( *(T (*)[]) first ), number( n ) {}

template< class T >
inline T* SubArray< T >::First() {
    return t; }
```

```
template< class T >
inline void SubArray< T >::Error( const char [] ) {
    abort(); }

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

Array

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

template< class T, int n >
inline Array< T, n >::Array() {}

template< class T, int n >
inline Array< T, n >::Array( SubArray< T > s ) {
    Copy( s ); }

template< class T, int n >
inline int Array< T, n >::Number() {
    return n; }

template< class T, int n >
inline T& Array< T, n >::operator []( int i ) {
    if (CHECKING && (i < 0 || i >= Number())) Error( "[]" );
    return a[ i ]; }

template< class T, int n >
inline SubArray< T > Array< T, n >::Sub( int start, int number ) {
    if (CHECKING && (start < 0 || start + number > Number()))
        Error( "Sub" );
    return SubArray< T >( First() + start, number ); }

template< class T, int n >
inline SubArray< T > Array< T, n >::Sub( int start ) {
    return Sub( start, Number() - start ); }

template< class T, int n >
inline SubArray< T > Array< T, n >::Sub() {
    return Sub( 0, Number() ); }

template< class T, int n >
inline Array< T, n >::operator SubArray< T >() {
    return Sub(); }

template< class T, int n >
inline Array< T, n >::Copy( SubArray< T > s ) {
    int n = Number();
    if (CHECKING && n != s.Number()) Error( "Copy" );
    T* to   = First();
    T* from = s.First();
    for (int i = 0; i < n; i++) *to++ = *from++; }

template< class T, int n >
inline int Array< T, n >::operator ==( SubArray< T > s ) {
    return Number() == s.Number() && First() == s.First(); }

template< class T, int n >
inline int Array< T, n >::operator !=( SubArray< T > s ) {
    return ! (*this == s); }

template< class T, int n >
inline int Array< T, n >::Equal( SubArray< T > s ) {
    int n = Number();
    if (n != s.Number()) return 0;
    T* to   = First();
    T* from = s.First();
```

```
    for (int i = 0; i < n; i++) if (*to++ != *from++) return 0;
    return 1; }

template< class T, int n >
inline T* Array< T, n >::First() {
    return a; }

template< class T, int n >
inline void Array< T,n >::Error( const char [] ) {
    abort(); }


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

DynArray

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

template< class T >
inline DynArray< T >::DynArray()
    : t( NULL ) {}

#ifdef NEWNEW
template< class T >
inline DynArray< T >::DynArray( int number )
    : t( new GCT[ number ] ) {}
#else
template< class T >
inline DynArray< T >::DynArray( int number ) {
    int bytes  = sizeof( int ) + sizeof( T ) * number;
    int* block = (int*) (new char[ bytes ]);
    *block     = number;
    t          = (T*) (block + 1); }
        /* This hack implementation doesn't run the constructor for T */
#endif

template< class T >
inline DynArray< T >::DynArray( SubArray< T > s ) {
    DynArray< T > d( s.Number() );
    t = d.t;
    Copy( s ); }

template< class T >
inline int DynArray< T >::Number() {
    return *(((int*) t) - 1);
        /* ...where new stores the size for gc arrays. */
    }

template< class T >
inline T& DynArray< T >::operator []( int i ) {
    if (CHECKING && (i < 0 || i >= Number())) Error( "[]" );
    return t[ i ]; }

template< class T >
inline SubArray< T > DynArray< T >::Sub( int start, int number ) {
    if (CHECKING && (start < 0 || start + number > Number()))
        Error( "Sub" );
    return SubArray< T >( First() + start, number ); }

template< class T >
inline SubArray< T > DynArray< T >::Sub( int start ) {
    return Sub( start, Number() - start ); }

template< class T >
inline SubArray< T > DynArray< T >::Sub() {
    return Sub( 0, Number() ); }
```

```
template< class T >
inline DynArray< T >::operator SubArray< T >() {
    return Sub(); }

template< class T >
inline DynArray< T > DynArray< T >::operator =( DynArray< T > d ) {
    t = d.t;
    return d; }

template< class T >
inline void DynArray< T >::Copy( SubArray< T > s ) {
    int n = Number();
    if (CHECKING && n != s.Number()) Error( "Copy" );
    T* to   = First();
    T* from = s.First();
    for (int i = 0; i < n; i++) *to++ = *from++; }

template< class T >
inline int DynArray< T >::operator ==( DynArray< T > d ) {
    return t == d.t; }

template< class T >
inline int DynArray< T >::operator ==( SubArray< T > s ) {
    return Number() == s.Number() && First() == s.First(); }

template< class T >
inline int DynArray< T >::operator !=( DynArray< T > d ) {
    return ! (*this == d); }

template< class T >
inline int DynArray< T >::operator !=( SubArray< T > s ) {
    return ! (*this == s); }

template< class T >
inline int DynArray< T >::Equal( SubArray< T > s ) {
    int n = Number();
    if (n != s.Number()) return 0;
    T* to   = First();
    T* from = s.First();
    for (int i = 0; i < n; i++) if (*to++ != *from++) return 0;
    return 1; }

template< class T >
inline T* DynArray< T >::First() {
    return t; }

template< class T >
inline void DynArray< T >::Error( const char [] ) {
    abort(); }

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

SUB macros

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

/* #define UNSAFE #pragma unsafe */

#define SUB( type, array ) \
    __SubArray< type, 1, 1, 1 >::New1( \
        array, (int)(sizeof( array ) / sizeof( array[ 0 ] )) )

#define SUB2( type, array, dim2 ) \
    __SubArray< type, dim2, 1, 1 >::New2( \
        array, (int)(sizeof( array ) / sizeof( array[ 0 ] )) )

#define SUB3( type, array, dim2, dim3 ) \
```

```
    __SubArray< type, dim2, dim3, 1 >::New3( \
        array, (int)(sizeof( array ) / sizeof( array[ 0 ] )) )

#define SUB4( type, array, dim2, dim3, dim4 ) \
    __SubArray< type, dim2, dim3, dim4 >::New4( \
        array, (int)(sizeof( array ) / sizeof( array[ 0 ] )) )

template< class T, int d2, int d3, int d4 >
class __SubArray {public:
    static SubArray< T > New1( T (&a)[], int n ) {
        return SubArray< T >( a, n ); };

    typedef Array< T, d2 > A2;
    static SubArray< A2 > New2( T (&a)[][d2], int n ) {
        return SubArray< A2 >( (A2*) a, n ); };

    typedef Array< Array< T, d3 >, d2 > A3;
    static SubArray< A3 > New3( T (&a)[][d2][d3], int n ) {
        return SubArray< A3 >( (A3*) a, n ); };

    typedef Array< Array< Array< T, d4 >, d3 >, d2 > A4;
    static SubArray< A4 > New4( T (&a)[][d2][d3][d4], int n ) {
        return SubArray< A4 >( (A4*) a, n ); };
    };

#endif /* _Array_h_ */
```

Appendix D: Text.h

This appendix presents the standard interface `Text.h`, which provides safe strings.

```
#ifndef _Text_h_
#define _Text_h_

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

Text and SubText

A `Text` is a heap-allocated varying-length string, and a `SubText` is a sub-sequence of a `Text`. `Text` and `SubText` are intended as safe replacements for most uses of `char*` and `string.h`.

A `Text` is a `DynArray<char>` and inherits all of the `DynArray` methods. A `SubText` is another name for `SubArray<char>`.

Many common operations can be performed using `DynArray` methods. For example, `t.Number()` gives the length of `Text t`, `t[i]` accesses the `i`-th character of `t`, and `t.Equal(t2)` compares `t` and `t2`.

The string-specific operations defined here operate on `SubTexts`. `DynArray` provides a default conversion from `Text` to `SubText`.

```
*********************************************************************/
#pragma safe

typedef SubArray< char > SubText;

class Text: public DynArray< char > {
public:

Text( const char* s );
      /* Returns a new Text that's a copy of s, with length strlen(s). */

Text( const char c );
      /* Returns a new Text of length 1 containing c. */

Text( SubText s );
      /* Returns a new Text that's a copy of s. */

friend Text operator +( SubText s1, SubText s2 );
      /* Returns a new Text containing the concatenation of s1 and s2. */

friend int EqualCase( SubText s1, SubText s2 );
      /* Returns true if s1 and s2 contain the same sequence of characters, ignoring case. */

friend int Compare( SubText s1, SubText s2, int ignoreCase = 0 );
      /* Returns -1, 0, or 1 depending on whether s1 is lexicographically less than, equal, or
      greater than s2. If ignoreCase, case is ignored in the comparison. */


friend int Find( SubText s1, char c, int ignoreCase = 0 );
friend int Find( SubText s1, SubText s2, int ignoreCase = 0 );
friend int FindRight( SubText s1, char c, int ignoreCase = 0 );
friend int FindRight( SubText s1, SubText s2, int ignoreCase = 0 );
      /* Returns the index of the leftmost (rightmost) occurrence of the character c or
      SubText s2 in the SubText s1. Returns -1 if the character or SubText is not
      found. If ignoreCase, case is ignored in the search. */
};
#endif /* _Text_h_ */
```

Appendix E: Variant.h

This appendix presents the standard interface `Variant.h`, which provides tagged unions.

```
#ifndef _Variant_h
#define _Variant_h

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

Variant

A `Variant` is a union combined with a tag, indicating which member the union currently contains. An attempt to access the incorrect member will (optionally) result in a checked runtime error. When using garbage collection, variants should be used in preference to unions wherever possible, since `Variants` avoid the collection inefficiencies caused by unions.

The template class Variant*n* creates a variant that can hold *n* different types. For example,

```
Variant2< int, char* > v;
```

declares `v` to be a variant that can contain an `int` or a `char*`. The `Variant`'s current tag value is accessed by:

```
v.Tag()
```

The tag values for a `Variant`*n* range from 0 to *n*-1, corresponding to the *n* types in the `Variant`*n*.

Assigning a value to a `Variant` automatically sets its tag:

```
v = 4;          // v.Tag() now equals 0
v = "Hello";    // v.Tag() now equals 1
```

The current value of a `Variant` can be accessed using explicit type conversion:

```
(char*) v
```

(`Variants` will also be implicitly converted whenever there's an unambiguous conversion.) Attempting to convert a `Variant` to a type other than that of the current value results in an (optional) checked runtime error.

`Variants` are standard classes that are handled specially by some compiler/collector implementations to avoid the collector inefficiencies associated with untagged unions. During a collection, these collectors use the `Variant`'s tag to determine the current type of its contents.

Note: Only `Variant2` is presented here. The other `Variant`*n* are almost identical.

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

#include "iostream.h"
#include "stdlib.h"

template< class T0, class T1 > class Variant2 {
public:
    int Tag();
       /* Returns the Variant's current tag, 0 if it's holding a T0, 1 if a T1. */

    void SetTag( int tag );
       /* Sets the Variant's current tag to tag, raising a checked runtime error if tag < 0
       or tag > 1. After the tag is changed, the contents of the Variant are undefined. */

    Variant2();
        /* Constructs a Variant with undefined tag and contents. */
```

```
    Variant2( const T0& t0 );
    Variant2( const T1& t1 );
       /* Constructs a Variant from a T0 or T1, setting the tag to 0 or 1 as appropriate. */

    operator T0&();
    operator T1&();
       /* A Variant can be converted to a T0 or a T1. Attempting to convert a variant to a T0
       when its tag != 0 raises a checked runtime error (similarly for T1). */

    Variant2< T0, T1 >& operator =( const Variant2< T0, T1 >& v );
       /* Assigns v to *this */

    Variant2< T0, T1 >& operator =( const T0& t0 );
    Variant2< T0, T1 >& operator =( const T1& t1 );
       /* Assigns a t0 or t1 to *this, setting its tag appropriately. */

private:
    int tag;
    union {T0 t0; T1 t1; } u;
    void Error( const char message[] ); };


/*********************************************************************

Inline implementation

*********************************************************************/

#ifndef CHECKING
#define CHECKING 1
#endif

template< class T0, class T1 >
inline int Variant2< T0, T1 >::Tag() {
    return tag; }

template< class T0, class T1 >
inline void Variant2< T0, T1 >::SetTag( int tag ) {
    if (CHECKING && (tag < 0 || tag > 1))
        Error( "Invalid Variant2 tag" );
    this->tag = tag; }

template< class T0, class T1 >
inline Variant2< T0, T1 >::Variant2(): tag( 0 ) {}

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

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

template< class T0, class T1 >
inline Variant2< T0, T1 >::operator T0&() {
        if (CHECKING && tag != 0) Error( "Variant2 has invalid tag" );
        return u.t0; }

template< class T0, class T1 >
inline Variant2< T0, T1 >::operator T1&() {
        if (CHECKING && tag != 1) Error( "Variant2 has invalid tag" );
        return u.t1; }

template< class T0, class T1 >
inline Variant2< T0, T1>&
Variant2< T0, T1 >::operator =( const Variant2< T0, T1 >& v ) {
```