Unsafe code

The core C# language, as defined in the preceding chapters, differs notably from C and C++ in its omission of pointers as a data type. Instead, C# provides references and the ability to create objects that are managed by a garbage collector. This design, coupled with other features, makes C# a much safer language than C or C++. In the core C# language it is simply not possible to have an uninitialized variable, a "dangling" pointer, or an expression that indexes an array beyond its bounds. Whole categories of bugs that routinely plague C and C++ programs are thus eliminated.

While practically every pointer type construct in C or C++ has a reference type counterpart in C#, nonetheless, there are situations where access to pointer types becomes a necessity. For example, interfacing with the underlying operating system, accessing a memory-mapped device, or implementing a time-critical algorithm may not be possible or practical without access to pointers. To address this need, C# provides the ability to write ***unsafe code***.

In unsafe code it is possible to declare and operate on pointers, to perform conversions between pointers and integral types, to take the address of variables, and so forth. In a sense, writing unsafe code is much like writing C code within a C# program.

Unsafe code is in fact a "safe" feature from the perspective of both developers and users. Unsafe code must be clearly marked with the modifier `unsafe`, so developers can't possibly use unsafe features accidentally, and the execution engine works to ensure that unsafe code cannot be executed in an untrusted environment.

Unsafe contexts

The unsafe features of C# are available only in unsafe contexts. An unsafe context is introduced by including an `unsafe` modifier in the declaration of a type or member, or by employing an *unsafe_statement*:

- A declaration of a class, struct, interface, or delegate may include an `unsafe` modifier, in which case the entire textual extent of that type declaration (including the body of the class, struct, or interface) is considered an unsafe context.
- A declaration of a field, method, property, event, indexer, operator, instance constructor, destructor, or static constructor may include an `unsafe` modifier, in which case the entire textual extent of that member declaration is considered an unsafe context.
- An *unsafe_statement* enables the use of an unsafe context within a *block*. The entire textual extent of the associated *block* is considered an unsafe context.

The associated grammar productions are shown below.

```
class_modifier_unsafe
    : 'unsafe'
    ;

struct_modifier_unsafe
    : 'unsafe'
    ;

interface_modifier_unsafe
    : 'unsafe'
    ;
```

```
delegate_modifier_unsafe
    : 'unsafe'
    ;

field_modifier_unsafe
    : 'unsafe'
    ;

method_modifier_unsafe
    : 'unsafe'
    ;

property_modifier_unsafe
    : 'unsafe'
    ;

event_modifier_unsafe
    : 'unsafe'
    ;

indexer_modifier_unsafe
    : 'unsafe'
    ;

operator_modifier_unsafe
    : 'unsafe'
    ;

constructor_modifier_unsafe
    : 'unsafe'
    ;

destructor_declaration_unsafe
    : attributes? 'extern'? 'unsafe'? '~' identifier '(' ')' destructor_body
    | attributes? 'unsafe'? 'extern'? '~' identifier '(' ')' destructor_body
    ;

static_constructor_modifiers_unsafe
    : 'extern'? 'unsafe'? 'static'
    | 'unsafe'? 'extern'? 'static'
    | 'extern'? 'static' 'unsafe'?
    | 'unsafe'? 'static' 'extern'?
    | 'static' 'extern'? 'unsafe'?
    | 'static' 'unsafe'? 'extern'?
    ;

embedded_statement_unsafe
    : unsafe_statement
    | fixed_statement
    ;

unsafe_statement
    : 'unsafe' block
    ;
```

In the example

```
public unsafe struct Node
{
    public int Value;
    public Node* Left;
    public Node* Right;
}
```

the `unsafe` modifier specified in the struct declaration causes the entire textual extent of the struct declaration to become an unsafe context. Thus, it is possible to declare the `Left` and `Right` fields to be of a pointer type. The example above could also be written

```
public struct Node
{
    public int Value;
    public unsafe Node* Left;
    public unsafe Node* Right;
}
```

Here, the `unsafe` modifiers in the field declarations cause those declarations to be considered unsafe contexts.

Other than establishing an unsafe context, thus permitting the use of pointer types, the `unsafe` modifier has no effect on a type or a member. In the example

```
public class A
{
    public unsafe virtual void F() {
        char* p;
        ...
    }
}

public class B: A
{
    public override void F() {
        base.F();
        ...
    }
}
```

the `unsafe` modifier on the `F` method in `A` simply causes the textual extent of `F` to become an unsafe context in which the unsafe features of the language can be used. In the override of `F` in `B`, there is no need to re-specify the `unsafe` modifier -- unless, of course, the `F` method in `B` itself needs access to unsafe features.

The situation is slightly different when a pointer type is part of the method's signature

```
public unsafe class A
{
    public virtual void F(char* p) {...}
}

public class B: A
{
    public unsafe override void F(char* p) {...}
}
```

Here, because `F` 's signature includes a pointer type, it can only be written in an unsafe context. However, the unsafe context can be introduced by either making the entire class unsafe, as is the case in `A` , or by including an `unsafe` modifier in the method declaration, as is the case in `B` .

Pointer types

In an unsafe context, a *type* (Types) may be a *pointer_type* as well as a *value_type* or a *reference_type*. However, a *pointer_type* may also be used in a `typeof` expression (Anonymous object creation expressions) outside of an unsafe context as such usage is not unsafe.

```
type_unsafe
    : pointer_type
    ;
```

A *pointer_type* is written as an *unmanaged_type* or the keyword `void` , followed by a `*` token:

```
pointer_type
    : unmanaged_type '*'
    | 'void' '*'
    ;

unmanaged_type
    : type
    ;
```

The type specified before the `*` in a pointer type is called the ***referent type*** of the pointer type. It represents the type of the variable to which a value of the pointer type points.

Unlike references (values of reference types), pointers are not tracked by the garbage collector -- the garbage collector has no knowledge of pointers and the data to which they point. For this reason a pointer is not permitted to point to a reference or to a struct that contains references, and the referent type of a pointer must be an *unmanaged_type*.

An *unmanaged_type* is any type that isn't a *reference_type* or constructed type, and doesn't contain *reference_type* or constructed type fields at any level of nesting. In other words, an *unmanaged_type* is one of the following:

- `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, or `bool`.
- Any *enum_type*.
- Any *pointer_type*.
- Any user-defined *struct_type* that is not a constructed type and contains fields of *unmanaged_type*s only.

The intuitive rule for mixing of pointers and references is that referents of references (objects) are permitted to contain pointers, but referents of pointers are not permitted to contain references.

Some examples of pointer types are given in the table below:

Example	Description
`byte*`	Pointer to `byte`
`char*`	Pointer to `char`
`int**`	Pointer to pointer to `int`
`int*[]`	Single-dimensional array of pointers to `int`
`void*`	Pointer to unknown type

For a given implementation, all pointer types must have the same size and representation.

Unlike C and C++, when multiple pointers are declared in the same declaration, in C# the `*` is written along with the underlying type only, not as a prefix punctuator on each pointer name. For example

```
int* pi, pj;    // NOT as int *pi, *pj;
```

The value of a pointer having type `T*` represents the address of a variable of type `T`. The pointer indirection operator `*` (Pointer indirection) may be used to access this variable. For example, given a variable `P` of type `int*`, the expression `*P` denotes the `int` variable found at the address contained in `P`.

Like an object reference, a pointer may be `null`. Applying the indirection operator to a `null` pointer results in implementation-defined behavior. A pointer with value `null` is represented by all-bits-zero.

The `void*` type represents a pointer to an unknown type. Because the referent type is unknown, the indirection operator cannot be applied to a pointer of type `void*`, nor can any arithmetic be performed on such a pointer. However, a pointer of type `void*` can be cast to any other pointer type (and vice versa).

Pointer types are a separate category of types. Unlike reference types and value types, pointer types do not inherit from `object` and no conversions exist between pointer types and `object`. In particular, boxing and unboxing (Boxing and unboxing) are not supported for pointers. However, conversions are permitted between different pointer types and between pointer types and the integral types. This is described in Pointer conversions.

A *pointer_type* cannot be used as a type argument (Constructed types), and type inference (Type inference) fails on generic method calls that would have inferred a type argument to be a pointer type.

A *pointer_type* may be used as the type of a volatile field (Volatile fields).

Although pointers can be passed as `ref` or `out` parameters, doing so can cause undefined behavior, since the pointer may well be set to point to a local variable which no longer exists when the called method returns, or the fixed object to which it used to point, is no longer fixed. For example:

```
using System;

class Test
{
    static int value = 20;

    unsafe static void F(out int* pi1, ref int* pi2) {
        int i = 10;
        pi1 = &i;

        fixed (int* pj = &value) {
            // ...
            pi2 = pj;
        }
    }
    static void Main() {
        int i = 10;
        unsafe {
            int* px1;
            int* px2 = &i;

            F(out px1, ref px2);

            Console.WriteLine("*px1 = {0}, *px2 = {1}",
                *px1, *px2);    // undefined behavior
        }
    }
}
```

A method can return a value of some type, and that type can be a pointer. For example, when given a pointer to a contiguous sequence of `int` s, that sequence's element count, and some other `int` value, the following method returns the address of that value in that sequence, if a match occurs; otherwise it returns `null` :

```
unsafe static int* Find(int* pi, int size, int value) {
    for (int i = 0; i < size; ++i) {
        if (*pi == value)
            return pi;
        ++pi;
    }
    return null;
}
```

In an unsafe context, several constructs are available for operating on pointers:

- The `*` operator may be used to perform pointer indirection (Pointer indirection).
- The `->` operator may be used to access a member of a struct through a pointer (Pointer member access).
- The `[]` operator may be used to index a pointer (Pointer element access).
- The `&` operator may be used to obtain the address of a variable (The address-of operator).
- The `++` and `--` operators may be used to increment and decrement pointers (Pointer increment and decrement).
- The `+` and `-` operators may be used to perform pointer arithmetic (Pointer arithmetic).
- The `==` , `!=` , `<` , `>` , `<=` , and `>=` operators may be used to compare pointers (Pointer comparison).
- The `stackalloc` operator may be used to allocate memory from the call stack (Fixed size buffers).

- The `fixed` statement may be used to temporarily fix a variable so its address can be obtained (The fixed statement).

Fixed and moveable variables

The address-of operator (The address-of operator) and the `fixed` statement (The fixed statement) divide variables into two categories: ***Fixed variables*** and ***moveable variables***.

Fixed variables reside in storage locations that are unaffected by operation of the garbage collector. (Examples of fixed variables include local variables, value parameters, and variables created by dereferencing pointers.) On the other hand, moveable variables reside in storage locations that are subject to relocation or disposal by the garbage collector. (Examples of moveable variables include fields in objects and elements of arrays.)

The `&` operator (The address-of operator) permits the address of a fixed variable to be obtained without restrictions. However, because a moveable variable is subject to relocation or disposal by the garbage collector, the address of a moveable variable can only be obtained using a `fixed` statement (The fixed statement), and that address remains valid only for the duration of that `fixed` statement.

In precise terms, a fixed variable is one of the following:

- A variable resulting from a *simple_name* (Simple names) that refers to a local variable or a value parameter, unless the variable is captured by an anonymous function.
- A variable resulting from a *member_access* (Member access) of the form `V.I`, where `V` is a fixed variable of a *struct_type*.
- A variable resulting from a *pointer_indirection_expression* (Pointer indirection) of the form `*P`, a *pointer_member_access* (Pointer member access) of the form `P->I`, or a *pointer_element_access* (Pointer element access) of the form `P[E]`.

All other variables are classified as moveable variables.

Note that a static field is classified as a moveable variable. Also note that a `ref` or `out` parameter is classified as a moveable variable, even if the argument given for the parameter is a fixed variable. Finally, note that a variable produced by dereferencing a pointer is always classified as a fixed variable.

Pointer conversions

In an unsafe context, the set of available implicit conversions (Implicit conversions) is extended to include the following implicit pointer conversions:

- From any *pointer_type* to the type `void*`.
- From the `null` literal to any *pointer_type*.

Additionally, in an unsafe context, the set of available explicit conversions (Explicit conversions) is extended to include the following explicit pointer conversions:

- From any *pointer_type* to any other *pointer_type*.
- From `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `ulong` to any *pointer_type*.
- From any *pointer_type* to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `ulong`.

Finally, in an unsafe context, the set of standard implicit conversions (Standard implicit conversions) includes the following pointer conversion:

- From any *pointer_type* to the type `void*`.

Conversions between two pointer types never change the actual pointer value. In other words, a conversion from one pointer type to another has no effect on the underlying address given by the pointer.

When one pointer type is converted to another, if the resulting pointer is not correctly aligned for the pointed-to type, the behavior is undefined if the result is dereferenced. In general, the concept "correctly aligned" is transitive: if a pointer to type `A` is correctly aligned for a pointer to type `B`, which, in turn, is correctly aligned for a pointer to type `C`, then a pointer to type `A` is correctly aligned for a pointer to type `C`.

Consider the following case in which a variable having one type is accessed via a pointer to a different type:

```
char c = 'A';
char* pc = &c;
void* pv = pc;
int* pi = (int*)pv;
int i = *pi;         // undefined
*pi = 123456;        // undefined
```

When a pointer type is converted to a pointer to byte, the result points to the lowest addressed byte of the variable. Successive increments of the result, up to the size of the variable, yield pointers to the remaining bytes of that variable. For example, the following method displays each of the eight bytes in a double as a hexadecimal value:

```
using System;

class Test
```

```
{
    unsafe static void Main() {
      double d = 123.456e23;
        unsafe {
           byte* pb = (byte*)&d;
            for (int i = 0; i < sizeof(double); ++i)
               Console.Write("{0:X2} ", *pb++);
            Console.WriteLine();
        }
    }
}
```

Of course, the output produced depends on endianness.

Mappings between pointers and integers are implementation-defined. However, on 32* and 64-bit CPU architectures with a linear address space, conversions of pointers to or from integral types typically behave exactly like conversions of `uint` or `ulong` values, respectively, to or from those integral types.

Pointer arrays

In an unsafe context, arrays of pointers can be constructed. Only some of the conversions that apply to other array types are allowed on pointer arrays:

- The implicit reference conversion (Implicit reference conversions) from any *array_type* to `System.Array` and the interfaces it implements also applies to pointer arrays. However, any attempt to access the array elements through `System.Array` or the interfaces it implements will result in an exception at run-time, as pointer types are not convertible to `object`.
- The implicit and explicit reference conversions (Implicit reference conversions, Explicit reference conversions) from a single-dimensional array type `S[]` to `System.Collections.Generic.IList<T>` and its generic base interfaces never apply to pointer arrays, since pointer types cannot be used as type arguments, and there are no conversions from pointer types to non-pointer types.
- The explicit reference conversion (Explicit reference conversions) from `System.Array` and the interfaces it implements to any *array_type* applies to pointer arrays.
- The explicit reference conversions (Explicit reference conversions) from `System.Collections.Generic.IList<S>` and its base interfaces to a single-dimensional array type `T[]` never applies to pointer arrays, since pointer types cannot be used as type arguments, and there are no conversions from pointer types to non-pointer types.

These restrictions mean that the expansion for the `foreach` statement over arrays described in The foreach statement cannot be applied to pointer arrays. Instead, a foreach statement of the form

```
foreach (V v in x) embedded_statement
```

where the type of `x` is an array type of the form `T[,,...,]`, `N` is the number of dimensions minus 1 and `T` or `V` is a pointer type, is expanded using nested for-loops as follows:

```
{
    T[,,...,] a = x;
    for (int i0 = a.GetLowerBound(0); i0 <= a.GetUpperBound(0); i0++)
    for (int i1 = a.GetLowerBound(1); i1 <= a.GetUpperBound(1); i1++)
    ...
    for (int iN = a.GetLowerBound(N); iN <= a.GetUpperBound(N); iN++) {
        V v = (V)a.GetValue(i0,i1,...,iN);
        embedded_statement
    }
}
```

The variables `a`, `i0`, `i1`, ..., `iN` are not visible to or accessible to `x` or the *embedded_statement* or any other source code of the program. The variable `v` is read-only in the embedded statement. If there is not an explicit conversion (Pointer conversions) from `T` (the element type) to `V`, an error is produced and no further steps are taken. If `x` has the value `null`, a `System.NullReferenceException` is thrown at run-time.

Pointers in expressions

In an unsafe context, an expression may yield a result of a pointer type, but outside an unsafe context it is a compile-time error for an expression to be of a pointer type. In precise terms, outside an unsafe context a compile-time error occurs if any *simple_name* (Simple names), *member_access* (Member access), *invocation_expression* (Invocation expressions), or *element_access* (Element access) is of a pointer type.

In an unsafe context, the *primary_no_array_creation_expression* (Primary expressions) and *unary_expression* (Unary operators) productions permit the following additional constructs:

```
primary_no_array_creation_expression_unsafe
    : pointer_member_access
    | pointer_element_access
    | sizeof_expression
    ;

unary_expression_unsafe
    : pointer_indirection_expression
    | addressof_expression
    ;
```

These constructs are described in the following sections. The precedence and associativity of the unsafe operators is implied by the grammar.

Pointer indirection

A *pointer_indirection_expression* consists of an asterisk (`*`) followed by a *unary_expression*.

```
pointer_indirection_expression
    : '*' unary_expression
    ;
```

The unary `*` operator denotes pointer indirection and is used to obtain the variable to which a pointer points. The result of evaluating `*P` , where `P` is an expression of a pointer type `T*` , is a variable of type `T` . It is a compile-time error to apply the unary `*` operator to an expression of type `void*` or to an expression that isn't of a pointer type.

The effect of applying the unary `*` operator to a `null` pointer is implementation-defined. In particular, there is no guarantee that this operation throws a `System.NullReferenceException` .

If an invalid value has been assigned to the pointer, the behavior of the unary `*` operator is undefined. Among the invalid values for dereferencing a pointer by the unary `*` operator are an address inappropriately aligned for the type pointed to (see example in Pointer conversions), and the address of a variable after the end of its lifetime.

For purposes of definite assignment analysis, a variable produced by evaluating an expression of the form `*P` is considered initially assigned (Initially assigned variables).

Pointer member access

A *pointer_member_access* consists of a *primary_expression*, followed by a " `->` " token, followed by an *identifier* and an optional *type_argument_list*.

```
pointer_member_access
    : primary_expression '->' identifier
    ;
```

In a pointer member access of the form `P->I` , `P` must be an expression of a pointer type other than `void*` , and `I` must denote an accessible member of the type to which `P` points.

A pointer member access of the form `P->I` is evaluated exactly as `(*P).I`. For a description of the pointer indirection operator (`*`), see Pointer indirection. For a description of the member access operator (`.`), see Member access.

In the example

```
using System;

struct Point
{
    public int x;
    public int y;

    public override string ToString() {
        return "(" + x + "," + y + ")";
    }
}

class Test
{
    static void Main() {
        Point point;
        unsafe {
            Point* p = &point;
            p->x = 10;
            p->y = 20;
            Console.WriteLine(p->ToString());
        }
    }
}
```

the `->` operator is used to access fields and invoke a method of a struct through a pointer. Because the operation `P->I` is precisely equivalent to `(*P).I`, the `Main` method could equally well have been written:

```
class Test
{
    static void Main() {
        Point point;
        unsafe {
            Point* p = &point;
            (*p).x = 10;
            (*p).y = 20;
            Console.WriteLine((*p).ToString());
        }
    }
}
```

Pointer element access

A *pointer_element_access* consists of a *primary_no_array_creation_expression* followed by an expression enclosed in " `[` " and " `]` ".

```
pointer_element_access
    : primary_no_array_creation_expression '[' expression ']'
    ;
```

In a pointer element access of the form `P[E]`, `P` must be an expression of a pointer type other than `void*`, and `E` must be an expression that can be implicitly converted to `int`, `uint`, `long`, or `ulong`.

A pointer element access of the form `P[E]` is evaluated exactly as `*(P + E)`. For a description of the pointer indirection operator (`*`), see Pointer indirection. For a description of the pointer addition operator (`+`), see Pointer arithmetic.

In the example

```
class Test
{
    static void Main() {
        unsafe {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) p[i] = (char)i;
        }
    }
}
```

a pointer element access is used to initialize the character buffer in a `for` loop. Because the operation `P[E]` is precisely equivalent to `*(P + E)`, the example could equally well have been written:

```
class Test
{
    static void Main() {
        unsafe {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) *(p + i) = (char)i;
        }
    }
}
```

The pointer element access operator does not check for out-of-bounds errors and the behavior when accessing an out-of-bounds element is undefined. This is the same as C and C++.

The address-of operator

An *addressof_expression* consists of an ampersand (`&`) followed by a *unary_expression*.

```
addressof_expression
    : '&' unary_expression
    ;
```

Given an expression `E` which is of a type `T` and is classified as a fixed variable (Fixed and moveable variables), the construct `&E` computes the address of the variable given by `E`. The type of the result is `T*` and is classified as a value. A compile-time error occurs if `E` is not classified as a variable, if `E` is classified as a read-only local variable, or if `E` denotes a moveable variable. In the last case, a fixed statement (The fixed statement) can be used to temporarily "fix" the variable before obtaining its address. As stated in Member access, outside an instance constructor or static constructor for a struct or class that defines a `readonly` field, that field is considered a value, not a variable. As such, its address cannot be taken. Similarly, the address of a constant cannot be taken.

The `&` operator does not require its argument to be definitely assigned, but following an `&` operation, the variable to which the operator is applied is considered definitely assigned in the execution path in which the operation occurs. It is the responsibility of the programmer to ensure that correct initialization of the variable actually does take place in this situation.

In the example

```
using System;

class Test
{
    static void Main() {
        int i;
        unsafe {
            int* p = &i;
            *p = 123;
        }
        Console.WriteLine(i);
    }
}
```

`i` is considered definitely assigned following the `&i` operation used to initialize `p`. The assignment to `*p` in effect initializes `i`, but the inclusion of this initialization is the responsibility of the programmer, and no compile-time error would occur if the assignment was removed.

The rules of definite assignment for the `&` operator exist such that redundant initialization of local variables can be avoided. For example, many external APIs take a pointer to a structure which is filled in by the API. Calls to such APIs typically pass the address of a local struct variable, and without the rule, redundant initialization of the struct variable would be required.

Pointer increment and decrement

In an unsafe context, the `++` and `--` operators (Postfix increment and decrement operators and Prefix increment and decrement operators) can be applied to pointer variables of all types except `void*`. Thus, for every pointer type `T*`, the following operators are implicitly defined:

```
T* operator ++(T* x);
T* operator --(T* x);
```

The operators produce the same results as `x + 1` and `x - 1`, respectively (Pointer arithmetic). In other words, for a pointer variable of type `T*`, the `++` operator adds `sizeof(T)` to the address contained in the variable, and the `--` operator subtracts `sizeof(T)` from the address contained in the variable.

If a pointer increment or decrement operation overflows the domain of the pointer type, the result is implementation-defined, but no exceptions are produced.

Pointer arithmetic

In an unsafe context, the `+` and `-` operators (Addition operator and Subtraction operator) can be applied to values of all pointer types except `void*`. Thus, for every pointer type `T*`, the following operators are implicitly defined:

```
T* operator +(T* x, int y);
T* operator +(T* x, uint y);
T* operator +(T* x, long y);
T* operator +(T* x, ulong y);

T* operator +(int x, T* y);
T* operator +(uint x, T* y);
T* operator +(long x, T* y);
T* operator +(ulong x, T* y);

T* operator -(T* x, int y);
T* operator -(T* x, uint y);
T* operator -(T* x, long y);
T* operator -(T* x, ulong y);

long operator -(T* x, T* y);
```

Given an expression `P` of a pointer type `T*` and an expression `N` of type `int`, `uint`, `long`, or `ulong`, the expressions `P + N` and `N + P` compute the pointer value of type `T*` that results from adding `N * sizeof(T)` to the address given by `P`. Likewise, the expression `P - N` computes the pointer value of type `T*` that results from subtracting `N * sizeof(T)` from the address given by `P`.

Given two expressions, `P` and `Q`, of a pointer type `T*`, the expression `P - Q` computes the difference between the addresses given by `P` and `Q` and then divides that difference by `sizeof(T)`. The type of the result is always `long`. In effect, `P - Q` is computed as `((long)(P) - (long)(Q)) / sizeof(T)`.

For example:

```
using System;

class Test
{
    static void Main() {
        unsafe {
            int* values = stackalloc int[20];
            int* p = &values[1];
            int* q = &values[15];
            Console.WriteLine("p - q = {0}", p - q);
            Console.WriteLine("q - p = {0}", q - p);
        }
    }
}
```

which produces the output:

```
p - q = -14
q - p = 14
```

If a pointer arithmetic operation overflows the domain of the pointer type, the result is truncated in an implementation-defined fashion, but no exceptions are produced.

Pointer comparison

In an unsafe context, the `==`, `!=`, `<`, `>`, `<=`, and `=>` operators (Relational and type-testing operators) can be applied to values of all pointer types. The pointer comparison operators are:

```
bool operator ==(void* x, void* y);
bool operator !=(void* x, void* y);
bool operator <(void* x, void* y);
bool operator >(void* x, void* y);
bool operator <=(void* x, void* y);
bool operator >=(void* x, void* y);
```

Because an implicit conversion exists from any pointer type to the `void*` type, operands of any pointer type can be compared using these operators. The comparison operators compare the addresses given by the two operands as if they were unsigned integers.

The sizeof operator

The `sizeof` operator returns the number of bytes occupied by a variable of a given type. The type specified as an operand to `sizeof` must be an *unmanaged_type* (Pointer types).

```
sizeof_expression
    : 'sizeof' '(' unmanaged_type ')'
    ;
```

The result of the `sizeof` operator is a value of type `int`. For certain predefined types, the `sizeof` operator yields a constant value as shown in the table below.

Expression	Result
`sizeof(sbyte)`	`1`
`sizeof(byte)`	`1`
`sizeof(short)`	`2`
`sizeof(ushort)`	`2`
`sizeof(int)`	`4`
`sizeof(uint)`	`4`
`sizeof(long)`	`8`
`sizeof(ulong)`	`8`
`sizeof(char)`	`2`
`sizeof(float)`	`4`
`sizeof(double)`	`8`
`sizeof(bool)`	`1`

For all other types, the result of the `sizeof` operator is implementation-defined and is classified as a value, not a constant.

The order in which members are packed into a struct is unspecified.

For alignment purposes, there may be unnamed padding at the beginning of a struct, within a struct, and at the end of the struct. The contents of the bits used as padding are indeterminate.

When applied to an operand that has struct type, the result is the total number of bytes in a variable of that type, including any padding.

The fixed statement

In an unsafe context, the *embedded_statement* (Statements) production permits an additional construct, the `fixed` statement, which is used to "fix" a moveable variable such that its address remains constant for the duration of the statement.

```
fixed_statement
    : 'fixed' '(' pointer_type fixed_pointer_declarators ')' embedded_statement
    ;

fixed_pointer_declarators
    : fixed_pointer_declarator (','  fixed_pointer_declarator)*
    ;

fixed_pointer_declarator
    : identifier '=' fixed_pointer_initializer
    ;

fixed_pointer_initializer
    : '&' variable_reference
    | expression
    ;
```

Each *fixed_pointer_declarator* declares a local variable of the given *pointer_type* and initializes that local variable with the address computed by the corresponding *fixed_pointer_initializer*. A local variable declared in a `fixed` statement is accessible in any *fixed_pointer_initializer*s occurring to the right of that variable's declaration, and in the *embedded_statement* of the `fixed` statement. A local variable declared by a `fixed` statement is considered read-only. A compile-time error occurs if the embedded statement attempts to modify this local variable (via assignment or the `++` and `--` operators) or pass it as a `ref` or `out` parameter.

A *fixed_pointer_initializer* can be one of the following:

- The token " `&` " followed by a *variable_reference* (Precise rules for determining definite assignment) to a moveable variable (Fixed and moveable variables) of an unmanaged type `T`, provided the type `T*` is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes the address of the given variable, and the variable is guaranteed to remain at a fixed address for the duration of the `fixed` statement.

- An expression of an *array_type* with elements of an unmanaged type `T`, provided the type `T*` is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes the address of the first element in the array, and the entire array is guaranteed to remain at a fixed address for the duration of the `fixed` statement. If the array expression is null or if the array has zero elements, the initializer computes an address equal to zero.
- An expression of type `string`, provided the type `char*` is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes the address of the first character in the string, and the entire string is guaranteed to remain at a fixed address for the duration of the `fixed` statement. The behavior of the `fixed` statement is implementation-defined if the string expression is null.
- A *simple_name* or *member_access* that references a fixed size buffer member of a moveable variable, provided the type of the fixed size buffer member is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes a pointer to the first element of the fixed size buffer (Fixed size buffers in expressions), and the fixed size buffer is guaranteed to remain at a fixed address for the duration of the `fixed` statement.

For each address computed by a *fixed_pointer_initializer* the `fixed` statement ensures that the variable referenced by the address is not subject to relocation or disposal by the garbage collector for the duration of the `fixed` statement. For example, if the address computed by a *fixed_pointer_initializer* references a field of an object or an element of an array instance, the `fixed` statement guarantees that the containing object instance is not relocated or disposed of during the lifetime of the statement.

It is the programmer's responsibility to ensure that pointers created by `fixed` statements do not survive beyond execution of those statements. For example, when pointers created by `fixed` statements are passed to external APIs, it is the programmer's responsibility to ensure that the APIs retain no memory of these pointers.

Fixed objects may cause fragmentation of the heap (because they can't be moved). For that reason, objects should be fixed only when absolutely necessary and then only for the shortest amount of time possible.

The example

```
class Test
{
    static int x;
    int y;
    unsafe static void F(int* p) {
        *p = 1;
    }

    static void Main() {
        Test t = new Test();
        int[] a = new int[10];
        unsafe {
            fixed (int* p = &x) F(p);
            fixed (int* p = &t.y) F(p);
            fixed (int* p = &a[0]) F(p);
            fixed (int* p = a) F(p);
        }
    }
}
```

demonstrates several uses of the `fixed` statement. The first statement fixes and obtains the address of a static field, the second statement fixes and obtains the address of an instance field, and the third statement fixes and obtains the address of an array element. In each case it would have been an error to use the regular `&` operator since the variables are all classified as moveable variables.

The fourth `fixed` statement in the example above produces a similar result to the third.

This example of the `fixed` statement uses `string`:

```
class Test
{
    static string name = "xx";

    unsafe static void F(char* p) {
        for (int i = 0; p[i] != '\0'; ++i)
            Console.WriteLine(p[i]);
    }

    static void Main() {
        unsafe {
            fixed (char* p = name) F(p);
            fixed (char* p = "xx") F(p);
        }
    }
}
```

In an unsafe context array elements of single-dimensional arrays are stored in increasing index order, starting with index `0` and ending with index `Length - 1`. For multi-dimensional arrays, array elements are stored such that the indices of the rightmost dimension are increased first, then the next left dimension, and so on to the left. Within a `fixed` statement that obtains a pointer `p` to an array instance `a`, the pointer values ranging from `p` to `p + a.Length - 1` represent addresses of the elements in the array. Likewise, the variables ranging from `p[0]` to `p[a.Length - 1]` represent the actual array elements. Given the way in which arrays are stored, we can treat an array of any dimension as though it were linear.

For example:

```
using System;

class Test
{
    static void Main() {
        int[,,] a = new int[2,3,4];
        unsafe {
            fixed (int* p = a) {
                for (int i = 0; i < a.Length; ++i)    // treat as linear
                    p[i] = i;
            }
        }

        for (int i = 0; i < 2; ++i)
            for (int j = 0; j < 3; ++j) {
                for (int k = 0; k < 4; ++k)
                    Console.Write("[{0},{1},{2}] = {3,2} ", i, j, k, a[i,j,k]);
                Console.WriteLine();
            }
    }
}
```

which produces the output:

```
[0,0,0] =  0 [0,0,1] =  1 [0,0,2] =  2 [0,0,3] =  3
[0,1,0] =  4 [0,1,1] =  5 [0,1,2] =  6 [0,1,3] =  7
[0,2,0] =  8 [0,2,1] =  9 [0,2,2] = 10 [0,2,3] = 11
[1,0,0] = 12 [1,0,1] = 13 [1,0,2] = 14 [1,0,3] = 15
[1,1,0] = 16 [1,1,1] = 17 [1,1,2] = 18 [1,1,3] = 19
[1,2,0] = 20 [1,2,1] = 21 [1,2,2] = 22 [1,2,3] = 23
```

In the example

```
class Test
{
    unsafe static void Fill(int* p, int count, int value) {
        for (; count != 0; count--) *p++ = value;
    }

    static void Main() {
        int[] a = new int[100];
        unsafe {
            fixed (int* p = a) Fill(p, 100, -1);
        }
    }
}
```

a `fixed` statement is used to fix an array so its address can be passed to a method that takes a pointer.

In the example:

```
unsafe struct Font
{
    public int size;
    public fixed char name[32];
}

class Test
{
    unsafe static void PutString(string s, char* buffer, int bufSize) {
        int len = s.Length;
        if (len > bufSize) len = bufSize;
        for (int i = 0; i < len; i++) buffer[i] = s[i];
        for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
    }

    Font f;

    unsafe static void Main()
    {
        Test test = new Test();
        test.f.size = 10;
        fixed (char* p = test.f.name) {
            PutString("Times New Roman", p, 32);
        }
    }
}
```

a fixed statement is used to fix a fixed size buffer of a struct so its address can be used as a pointer.

A `char*` value produced by fixing a string instance always points to a null-terminated string. Within a fixed statement that obtains a pointer `p` to a string instance `s`, the pointer values ranging from `p` to `p + s.Length - 1` represent addresses of the characters in the string, and the pointer value `p + s.Length` always points to a null character (the character with value `'\0'`).

Modifying objects of managed type through fixed pointers can results in undefined behavior. For example, because strings are immutable, it is the programmer's responsibility to ensure that the characters referenced by a pointer to a fixed string are not modified.

The automatic null-termination of strings is particularly convenient when calling external APIs that expect "C-style" strings. Note, however, that a string instance is permitted to contain null characters. If such null characters are present, the string will appear truncated when treated as a null-terminated `char*`.

Fixed size buffers

Fixed size buffers are used to declare "C style" in-line arrays as members of structs, and are primarily useful for interfacing with unmanaged APIs.

Fixed size buffer declarations

A ***fixed size buffer*** is a member that represents storage for a fixed length buffer of variables of a given type. A fixed size buffer declaration introduces one or more fixed size buffers of a given element type. Fixed size buffers are only permitted in struct declarations and can only occur in unsafe contexts (Unsafe contexts).

```
struct_member_declaration_unsafe
    : fixed_size_buffer_declaration
    ;

fixed_size_buffer_declaration
    : attributes? fixed_size_buffer_modifier* 'fixed' buffer_element_type fixed_s
    ;

fixed_size_buffer_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'unsafe'
    ;

buffer_element_type
    : type
    ;
```

```
fixed_size_buffer_declarator
    : identifier '[' constant_expression ']'
    ;
```

A fixed size buffer declaration may include a set of attributes (Attributes), a `new` modifier (Modifiers), a valid combination of the four access modifiers (Type parameters and constraints) and an `unsafe` modifier (Unsafe contexts). The attributes and modifiers apply to all of the members declared by the fixed size buffer declaration. It is an error for the same modifier to appear multiple times in a fixed size buffer declaration.

A fixed size buffer declaration is not permitted to include the `static` modifier.

The buffer element type of a fixed size buffer declaration specifies the element type of the buffer(s) introduced by the declaration. The buffer element type must be one of the predefined types `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, or `bool`.

The buffer element type is followed by a list of fixed size buffer declarators, each of which introduces a new member. A fixed size buffer declarator consists of an identifier that names the member, followed by a constant expression enclosed in `[` and `]` tokens. The constant expression denotes the number of elements in the member introduced by that fixed size buffer declarator. The type of the constant expression must be implicitly convertible to type `int`, and the value must be a non-zero positive integer.

The elements of a fixed size buffer are guaranteed to be laid out sequentially in memory.

A fixed size buffer declaration that declares multiple fixed size buffers is equivalent to multiple declarations of a single fixed size buffer declaration with the same attributes, and element types. For example

```
unsafe struct A
{
   public fixed int x[5], y[10], z[100];
}
```

is equivalent to

```
unsafe struct A
{
   public fixed int x[5];
   public fixed int y[10];
```

```
    public fixed int z[100];
}
```

Fixed size buffers in expressions

Member lookup (Operators) of a fixed size buffer member proceeds exactly like member lookup of a field.

A fixed size buffer can be referenced in an expression using a *simple_name* (Type inference) or a *member_access* (Compile-time checking of dynamic overload resolution).

When a fixed size buffer member is referenced as a simple name, the effect is the same as a member access of the form `this.I`, where `I` is the fixed size buffer member.

In a member access of the form `E.I`, if `E` is of a struct type and a member lookup of `I` in that struct type identifies a fixed size member, then `E.I` is evaluated an classified as follows:

- If the expression `E.I` does not occur in an unsafe context, a compile-time error occurs.
- If `E` is classified as a value, a compile-time error occurs.
- Otherwise, if `E` is a moveable variable (Fixed and moveable variables) and the expression `E.I` is not a *fixed_pointer_initializer* (The fixed statement), a compile-time error occurs.
- Otherwise, `E` references a fixed variable and the result of the expression is a pointer to the first element of the fixed size buffer member `I` in `E`. The result is of type `S*`, where `S` is the element type of `I`, and is classified as a value.

The subsequent elements of the fixed size buffer can be accessed using pointer operations from the first element. Unlike access to arrays, access to the elements of a fixed size buffer is an unsafe operation and is not range checked.

The following example declares and uses a struct with a fixed size buffer member.

```
unsafe struct Font
{
    public int size;
    public fixed char name[32];
}

class Test
{
    unsafe static void PutString(string s, char* buffer, int bufSize) {
        int len = s.Length;
        if (len > bufSize) len = bufSize;
        for (int i = 0; i < len; i++) buffer[i] = s[i];
        for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
    }

    unsafe static void Main()
    {
        Font f;
        f.size = 10;
        PutString("Times New Roman", f.name, 32);
    }
}
```

Definite assignment checking

Fixed size buffers are not subject to definite assignment checking (Definite assignment), and fixed size buffer members are ignored for purposes of definite assignment checking of struct type variables.

When the outermost containing struct variable of a fixed size buffer member is a static variable, an instance variable of a class instance, or an array element, the elements of the fixed size buffer are automatically initialized to their default values (Default values). In all other cases, the initial content of a fixed size buffer is undefined.

Stack allocation

In an unsafe context, a local variable declaration (Local variable declarations) may include a stack allocation initializer which allocates memory from the call stack.

```
local_variable_initializer_unsafe
    : stackalloc_initializer
    ;

stackalloc_initializer
    : 'stackalloc' unmanaged_type '[' expression ']'
    ;
```

The *unmanaged_type* indicates the type of the items that will be stored in the newly allocated location, and the *expression* indicates the number of these items. Taken together, these specify the required allocation size. Since the size of a stack allocation cannot be negative, it is a compile-time error to specify the number of items as a *constant_expression* that evaluates to a negative value.

A stack allocation initializer of the form `stackalloc T[E]` requires `T` to be an unmanaged type (Pointer types) and `E` to be an expression of type `int`. The construct allocates `E * sizeof(T)` bytes from the call stack and returns a pointer, of type `T*`, to the newly allocated block. If `E` is a negative value, then the behavior is undefined. If `E` is zero, then no allocation is made, and the pointer returned is implementation-defined. If there is not enough memory available to allocate a block of the given size, a `System.StackOverflowException` is thrown.

The content of the newly allocated memory is undefined.

Stack allocation initializers are not permitted in `catch` or `finally` blocks (The try statement).

There is no way to explicitly free memory allocated using `stackalloc`. All stack allocated memory blocks created during the execution of a function member are automatically discarded when that function member returns. This corresponds to the `alloca` function, an extension commonly found in C and C++ implementations.

In the example

```
using System;

class Test
{
    static string IntToString(int value) {
        int n = value >= 0? value: -value;
        unsafe {
            char* buffer = stackalloc char[16];
            char* p = buffer + 16;
            do {
                *--p = (char)(n % 10 + '0');
                n /= 10;
            } while (n != 0);
            if (value < 0) *--p = '-';
            return new string(p, 0, (int)(buffer + 16 - p));
        }
    }

    static void Main() {
        Console.WriteLine(IntToString(12345));
        Console.WriteLine(IntToString(-999));
    }
}
```

a `stackalloc` initializer is used in the `IntToString` method to allocate a buffer of 16 characters on the stack. The buffer is automatically discarded when the method returns.

Dynamic memory allocation

Except for the `stackalloc` operator, C# provides no predefined constructs for managing non-garbage collected memory. Such services are typically provided by supporting class libraries or imported directly from the underlying operating system. For example, the `Memory` class below illustrates how the heap functions of an underlying operating system might be accessed from C#:

```
using System;
using System.Runtime.InteropServices;

public static unsafe class Memory
{
    // Handle for the process heap. This handle is used in all calls to the
    // HeapXXX APIs in the methods below.
    private static readonly IntPtr s_heap = GetProcessHeap();

    // Allocates a memory block of the given size. The allocated memory is
    // automatically initialized to zero.
    public static void* Alloc(int size)
    {
        void* result = HeapAlloc(s_heap, HEAP_ZERO_MEMORY, (UIntPtr)size);
        if (result == null) throw new OutOfMemoryException();
        return result;
    }

    // Copies count bytes from src to dst. The source and destination
    // blocks are permitted to overlap.
    public static void Copy(void* src, void* dst, int count)
    {
        byte* ps = (byte*)src;
        byte* pd = (byte*)dst;
        if (ps > pd)
        {
            for (; count != 0; count--) *pd++ = *ps++;
        }
        else if (ps < pd)
        {
            for (ps += count, pd += count; count != 0; count--) *--pd = *--ps;
        }
    }

    // Frees a memory block.
    public static void Free(void* block)
    {
        if (!HeapFree(s_heap, 0, block)) throw new InvalidOperationException();
    }

    // Re-allocates a memory block. If the reallocation request is for a
    // larger size, the additional region of memory is automatically
    // initialized to zero.
    public static void* ReAlloc(void* block, int size)
    {
        void* result = HeapReAlloc(s_heap, HEAP_ZERO_MEMORY, block, (UIntPtr)size
        if (result == null) throw new OutOfMemoryException();
        return result;
    }

    // Returns the size of a memory block.
    public static int SizeOf(void* block)
    {
        int result = (int)HeapSize(s_heap, 0, block);
        if (result == -1) throw new InvalidOperationException();
        return result;
    }

    // Heap API flags
    private const int HEAP_ZERO_MEMORY = 0x00000008;

    // Heap API functions
    [DllImport("kernel32")]
    private static extern IntPtr GetProcessHeap();

    [DllImport("kernel32")]
    private static extern void* HeapAlloc(IntPtr hHeap, int flags, UIntPtr size);

    [DllImport("kernel32")]
    private static extern bool HeapFree(IntPtr hHeap, int flags, void* block);

    [DllImport("kernel32")]
    private static extern void* HeapReAlloc(IntPtr hHeap, int flags, void* block,

    [DllImport("kernel32")]
    private static extern UIntPtr HeapSize(IntPtr hHeap, int flags, void* block);
}
```

An example that uses the `Memory` class is given below:

```
class Test
{
    static unsafe void Main()
    {
        byte* buffer = null;
        try
        {
            const int Size = 256;
            buffer = (byte*)Memory.Alloc(Size);
            for (int i = 0; i < Size; i++) buffer[i] = (byte)i;
            byte[] array = new byte[Size];
            fixed (byte* p = array) Memory.Copy(buffer, p, Size);
            for (int i = 0; i < Size; i++) Console.WriteLine(array[i]);
        }
        finally
        {
            if (buffer != null) Memory.Free(buffer);
        }
        }
    }
```

The example allocates 256 bytes of memory through `Memory.Alloc` and initializes the memory block with values increasing from 0 to 255. It then allocates a 256 element byte array and uses `Memory.Copy` to copy the contents of the memory block into the byte array. Finally, the memory block is freed using `Memory.Free` and the contents of the byte array are output on the console.