Chapter 1

Language Defined Types

A type specifies how to interpret a sequence of bits and the operations that can be applied to those bits.

1.1 Fundamental Types

A fundamental type is a type that cannot be described in terms of other types. The `cweet` language provides the following fundamental types.

- `void` only value is void
- `bool`
- `short`, `int`, `long`, `llong` and sized types `i8`, `i16`, `i32`, `i64`, `i128` signed arithmetic
- `ushort`, `uint`, `ulong`, `ullong` for modulo arithmetic and their sized counterparts `u8`, `u16`, `u32`, `u64`, `u128`
- `byte`, `word`, `b8`, `b16`, `b32`, `b64`, `b128` bitwise operations
- `char`, `wchar`, `c8`, `c16`, `c32` characters
- `float`, `double`, `ldouble`, `ddouble`, `f8`, `f16`, `f32`, `f64`, `f128` floating point

The result of overflow or underflow on signed integers is implementation defined. Use `uint` for modulo arithmetic. Use sized types only when there is a special need for it. Use `double` when a floating-point number is needed.

1.2 Complex Types

A complex type is a type that is defined in terms of other types. More advanced programming languages provide facilities to define complex types called generic types. In `cweet` the following are the only complex types.

- `[n]T` An array of `n` elements of type `T`.

- `[]T` A slice of elements of type `T`.
- `&T` A non-nullable pointer to an object of type `T`.
- `*T` A nullable pointer to an object of type `T`.
- **maybe**`!T` A value of type `T` or nothing.

1.2.1 Arrays

An array `[n]T` consists of `n` elements of type `T` laid out contiguously in memory. If `T` is an array type, then the type `T` is called a multi-dimensional array. A type `[a1][a2]...[ak]T` where `T` is not an array type is a `k`-dimensional array with dimensions `a1,...,ak` and base type `T`. It can also be viewed as a 1-dimensional array of dimension `a1` where each element is of type `[a2]...[ak]T`.

Arrays support assignment, indexing, and slicing.

If `a : [a(1)]...[a(k)]T`, then an index expression `a[i(1)]...[i(m)]` refers to the element of type `[a(m+1)]...[a(k)]T` allocated as part of `a` where `m <= k` and `0 <= i(j) < a(j)` for all `j`.

The expression `a[]` produces a `k`-dimensional slice with the same dimensions as `a` that refers to the elements of `a`. The expression `a[i..j]` produces a `k`-dimensional slice that refers to rows `i` through `j` of the array `a`.

1.2.2 Slices

A `k`-dimensional slice with base type `T` is the following type.

```
struct {
    ptr : &T,
    len : [k]int
}
```

and refers to a contiguous block of memory holding elements of type `T`.

A `k`-dimensional slice can be sliced further to create another `k`-dimensional slice. The expression `s[i(1)]...[i(m)]` produces an `k-m` dimensional slice obtained by fixing the first `m` indices to these values. If `m = k`, this expression refers to the underlying value. The expression `s[i..j]` produces a `k`-dimensional slice that refers to the contiguous block of memory referred to by `s[i]` through `s[j]`.

The following function in `std::slice` can be used to reshape slices.

```
fn reshape (
        in : slice!(n, T),
        dims : [m]int
   ) : slice!(m, T)
    where (n, m : int, T : Type);
```

This produces an `m` dimensional slice from an `n` dimensional slice. The total number of elements have to be the same.

Slices of slices can be used to create jagged arrays. Unlike multidimensional slices, the base elements need not be contiguously laid out in memory. For example, the type `[]([]int)` is a slice of slices of `int`s.

1.2.3 Pointers

The type `&T` can never be `Null`. This allows the compiler to elide null pointer checks.

The type `*T` can take `Null` as a value and therefore the expression `*a` can fail. This type can be deconstructed in two ways: 1) As a **choice** between `Val{T}` and `Null` or between `Ptr{&T}` and `Null`.

1.2.4 Maybe

An object of type **maybe**`!T` can optionally hold a value of type `T`. This type is used to represent expressions producing values of type `T` that can fail. It can be deconstructed as a **choice** between `Just{T}` and `Nothing`.

1.3 User Defined Types

The **type** keyword is used to define types.

An **enum** is a restriction of values of one of the integral types.

```
type Colour = enum {
  red,
  green,
  blue
};
// Use integral values for constants. Like C enum.
type Colour = enum(int) {
  red = 1,
  green,
  blue
};
// Use a different integral type.
type Colour = enum(long) {
  red, green, blue,
  white, black, gray,
};
```

enum types are not interchangeable with integral types unless specified in the declaration. By the same rule, you cannot specify values of enum members as integers unless you specify a base integral type. An enum with a base integral type has all the operations of the base integral type.

The **struct** keyword is used to define product types. All members of a struct are labelled. The layout is the same as a **struct** in `C`.

```
type Point2D = struct { x : int, y : int };
```

A **struct** with unlabelled members is called a tuple. The members are implicitly labelled `a, b, ....`

```
type Point3D = {int, int, int};
// equivalent to
type Point3D = struct {
    a : int,
    b : int,
    c : int
};
```

The **union** keyword is the same as `C` **union**.

The **choice** keyword can be used to define tagged or discriminative unions. A **choice** has a finite number of choices and data members in each of those choices.

```
type NetStatus = choice {
  Connected{ip : [4]byte},
  Disconnected
};
```

The data members in a **choice** (For example, `ip`) can be left unnamed.

An anonymous discriminative union is the sum analogue of tuples.

```
type IoF = int | float;
```

Any value of type present in an unlabelled choice can be implicitly converted to a value of the unlabelled choice type. For example, it is valid to initialize or assign an object of type `IoF` from an `int` or a `float`.

Anonymous sums have the following properties.

$$\begin{aligned} (absorption) X|X &= X \\ (commutativity) X|Y &= Y|X \\ (associativity) X|(Y|Z) &= (X|Y)|Z \end{aligned}$$

A type definition **type** `T = U` creates a wrapper type `T` for `U`. Objects of type `T` and `U` can be explicitly converted to each other.

The `alias` keyword can be used to give new names to existing symbols.

1.4 Aggregate Types

Arrays, slices, maybe, structs, tuples, unions, and tagged and anonymous choice types are all aggregate types. Each aggregate type is a collection of member types. For example, for slices, the members are a pointer to the underlying type and an `int`.

1.5 Type Conversions

The standard module `std::conv` provides functions that implement explicit type conversions supported by the language.

TODO: Define allowed conversions.

1.6 Subtyping

We denote the subtyping relation by <:. This relation is reflexive and transitive. For pointers to fundamental types, we have the following subtyping relationships.

1. If `S` is the same as type `T` except for stricter alignment, then `S` <: `T`.
2. The type constructors `array, slice, maybe, pointer, npointer` are covariant. Additionally, we have `&X` <: `*Y` if `X` <: `Y`.

1.7 Implicit Conversions

We write `X` → `Y` for types `X` and `Y` to specify that `X` is implicitly convertible to `Y`.

1. `X` <: `Y` implies `X` → `Y`.
2. `X` → `(X'|Y)` if `X` → `X'` and not `X` → `Y`.
3. `X|Y` → `X'` if `X` → `X'` and `Y` → `X'`.
4. Enums with integral types `I` are implicitly convertible to `I`.
5. For error-full types with success value of type `T`, `T` is implicitly convertible to the error-full type.
6. **error** is implicitly convertible to any error type.

Chapter 2

Statements and Expressions

A statement produces no useful value and is specified for potential side-effects. An expression produces a value that can be used by other expressions or statements. In `cweet` many constructs are available as both statements and expressions. For example:

```
// if-else expression.
var max = if (a > b) a else b;

// if-else statement.
var max : int = undefined;

if (a > b) max = a;
else max = b;
```

On the other hand, constructs such as **return** statements and assignment statements are not available as expressions.

2.1 Variable Declarations

A variable declaration is a statement such as:

```
var i : int = 42;
```

does the following things:

1. Allocates storage for an `int`.
2. Binds the identifier `i` to the storage.
3. Initializes the storage location with the representation for value `42`.

For user-defined types, one can define variables as follows:

```
// For product types use the typename
var p = Point2D{0, 0};
var q = Point2D{.x = 1, .y = 1};
```

```
var r = Point3D{0, 0, 0};

// For tuples, use a tuple literal.
var x = {0, 1};

// For choice types use the constructors.
var s = NetStatus::Connected{[127, 0, 0, 1]};

// For unlabelled choices, infer constructor.
var x : IoF = 42; // int

// For enums, use one of the members.
var bg = black;
var fg = Colour::white;
var border : Colour = red;
```

When defining variables, an `_` can be used to ask the compiler to infer parts of the type specification.

The special value **undefined** can be used to leave a storage location uninitialized. An `_` can be used in place of the variable name if the variable is not going to be used.

2.2 Literals

2.2.1 Numeric Literals

Integral literals are polymorphic and default to `int`. Floating point literals are also polymorphic and default to `double`. Underscore can be used as a digit separator. The prefixes `0xX` and `0bB` can be used for hexadecimal and binary format. The character `p` can be used for scientific notation.

```
100_000
0xcad
0b101010
6.023p23
1p-6
0b1.110p+3
```

A numeric literal cannot be followed by an alphanumeric, underscore, or a period.

2.2.2 String Literals

String literals are stored as a null-terminated array of `char` in the data section. Their type is `[]char` in `cweet` and the null-terminator is ignored by the slice.

A string enclosed within " is a string literal with escape sequences. A raw string literal is written as r{c<Contents here>c}. The character c can be any character except } and acts as part of the delimiter.

```
"Hello, world\n"
"utf8 \u8{0d09}"
"Embedded null bytes \x00\x00"
r{ Hello, world } // space as delimiter.
r{|
   int main()
   {
       return 0;
   }
|} // Everything between two |'s.
```

2.2.3 User Defined Type Literals

```
Point{.x = 9, .y = 1} // struct
{.x = 9, .y = 1} // struct, type inferred
Network::Connected{[127, 0, 0, 1]} // choice
Connected{[127, 0, 0, 1]} // choice, type inferred.
Network::Disconnected // choice
Colour::red // enum
red // enum, type inferred
```

2.2.4 Symbols

A symbol literal is an identifier preceeded by a backquote. A symbol stands for itself and symbols are available only at compile-time. These are used as **loop** labels and for introducing variable bindings.

2.3 Assignment

An assignment statement mutates some storage location. The simplest form `lexpr = expr;` mutates the contents of the storage location referred to by `lexpr` using the value of `expr`. The assignment operation is defined for all types as a simple `memcpy`.

For infix operators `op` except for comparison operators, the augmented assignment `a op= b;` is same as in `C`.

2.4 Comparison Operations

The operators `==`, `=/=`, `<`, `>`, `>=`, `<=` are available. They produce Boolean values. The equality operators are defined for all fundamental types. They are not defined for user defined types because of padding. The comparison operators are defined for all numerical types includine `byte` etc.

2.5 Arithmetic Operations

Usual binary arithmetic operators `+, -, *, /, %` are supported. The unary `-` evaluates to the additive inverse in the ring. The unary `+` is identity and is provided for consistency.

2.6 Boolean Logic

The operators `&&, ||, !` are used to combine Boolean expressions. In addition to comparison operations, the following expressions also produce Boolean values.

```
type Colour = enum { red, green, blue };
var bg = red;
assert (bg.red && !bg.green);
```

2.7 Conditionals

A conditional can be an expression or a statement.

If `expr` has type `T`, then **when**`(condn) expr` has type `T|void`. In the statement **when** `(expr) stmt`, `stmt` is executed only if `expr` evaluates to `true`.

If `expr1` and `expr2` has types `T` and `U`, then **if**`(condn) expr1` **else** `expr2` has type `(T|U)`.

An **ensure** produces **maybe!**`T`.

```
(ensure (x != 0) 1/x) : maybe!double
```

Similarly, a guard expression has type that is the anonymous discriminative union of its arms' types. If the catch-all pattern `_` is absent, then the type union contains `void`.

```
fn abs(n : int) : int
{
    return guard {
        n >= 0 => n,
        _      => -n,
    };
}
```

Therefore, the following fails to compile.

```
fn abs(n : int) : int
{
    return guard {
        n >= 0 => n,
        n < 0  => -n,
    };
}
```

The values **unreachable** and **error** have types **unreachable** and **error**. The following demonstrates how to use these with **guard** expressions.

```
return guard {
    n > 0 => n,
    _     => -n,
} : int

return guard {
    n > 0 => n,
    n < 0 => -n,
} : int|void

return guard {
    n > 0 => n,
    n < 0 => -n,
    _     => unreachable,
} : int|unreachable -> int

return guard {
    n > 0 => n,
    n < 0 => -n,
    _     => error,
} : int|error -> maybe!int
```

2.8 Loops

The **loop** expression and statement is inspired by Lisp's loop macro. The general form is:

```
loop
'label
for (var x; slice)
while (bexpr1)
until (bexpr2)
break (val)
stmt
```

The **with** clause is used to setup variables only used within loop's scope. The **for** clause specifies iteration variables and what to iterate over. The **until** and **while** clauses specify stopping criterion. If the loop is an expresion, then the **break** clause specifies the value of the expression. The body of the loop may also contain **break** statements, optionally labelled, that may or may not return a value. The body of the loop must be enclosed in braces.

```
var i = loop for(var i; 1000..)
            until (is_prime(i)) break(i) {};
```

assigns the smallest prime larger than `1000` to `i`.

A **continue** statement can be used to skip to the next iteration of the loop. A label can be specified to identify the loop.

2.9 Lexical scoping

The **block** keyword can be used to specify that a block of code delimited by braces is an expression. The type of this expression is the type of the last expression in the block. (This keyword is used to keep the grammar LL(1) by avoiding conflicts with tuple/struct literals).

```
var i = block {
    // Complex initialization.
    e // expression used to initialize i
};
```

A statement beginning with an open brace is treated as the start of a lexical scope.

```
{
    var a = 0;
    print(a);
}

{0, 0}; // Error.
({0, 0}); // Ok
```

On the other hand, when expecting an expression, the compiler treats an open brace as the beginning of a struct literal. The **block** keyword can be used to specify that the following open brace indicates the start of a block expression and not a struct literal.

The **with** keyword introduces a new lexical block along with some statements at the beginning of the block. This generalizes initializing expressions in for-loops in `C` and conditional statements in `C++`. The value of the following expression is the last element of the list `head`.

```
with (
    var p = head;
    var q = head;
) loop while (p != Null) break(q) {
    q = p;
    p = p.next;
}
```

A **with**`(...)` can be followed by a statement, expression (If **with** is an expression), or a sequence of statements enclosed between braces.

2.10 Dereferencing Operations

Pointers can be dereferenced using the unary prefix operator `*`. A `*T` dereferences into a **maybe!**`T` and a `&T` dereferences into a `T`.

2.11 Destructuring Operations

A **match** statement can be used to pattern match against many types.

```
// int
match (i) {
    `j == .. 100 => j,
    _            => @max(:int),
}

// arrays
match (a) {
    [`i == 0..100, ...]       => i,
    [_, `j == 0 .. 100, ...] => j,
    _                         => 1000
}

// enum
match (day) {
    // range of values.
    Tuesday .. Thursday => "Midweek",
    // set of values.
    {Monday, Friday}    => "Bridgeday",
    _                   => "Weekend",
}

// choice
match (status) {
    Connected{[127, 0, 0, 1]}   => "localhost",
    Connected{[255, 255, ...]}  => "localnet",
    Connected{`a .ip}           => format_ip(a),
    Disconnected                => "disconnected"
}

// tuple
match (point) {
    {0, 0}       => origin,
    {0..5, 0..5} => near_origin,
    _            => somewhere
}
```

```
// struct
match (point) {
    // Use field names.
    {.y == 0, .x == 0} => origin,
    {0..5, 0..5}       => near_origin,
    _                  => somewhere
}
```

The general form of a pattern to match against a field is

```
'id .field :type == pat
```

This binds the named field to the identifier `id` and the pattern succeeds only if value of `field` matches `pat`. The **type** is used when matching against anonymous choices.

The type of a **match** expression is assigned using the same rules as for **guard**.

A **using** statement can be used to expose the fields of structs.

```
// point : {int, int}
var dist = using (point) sqrt(a*a + b*b);
```

It can also be used to unwrap specific choices.

```
var r = using (status.Connected) block {
    // Use ip : [4]byte here.
};
```

This type of **using** can be helpful when one wants to unwrap deeply nested choice types. The expression `status.Connected` has type **maybe!**`S` where `S` is a compiler generated struct type **struct** `{ ip : [4]byte }`. An **is** expression can be used to select a particular choice from an anonymous discriminative union.

```
var x : (int|bool) = 42;
var y = (x is int) + 1;
```

Here the expression `x` **is** `int` has type **maybe!**`int`.

The following form can be used to bind names in a **using** statement.

```
var r = using (
    position as 'p,
    status.Connected as 's
) block {
    // p.x, p.y, s.ip
};
```

Notice that the **using** statement allows treating computations that could fail as if they do not. This is the primary purpose of **using**.

In general, a **using** `(expr as pat) { }` executes the block of code as if `expr` evaluated without error and the value of `expr` successfully matched with `pat`. If the block of code could fail, then the **using** expression may also fail.

2.12 Array and Slice Operations

Arrays and slices have member `len : int` that gives the number of elements in the array. A slice can be obtained from an array or a slice using a slice expression.

```
// primes has type [6]int
var primes = [2, 3, 5, 7, 11, 13];
var a = primes[];
var b = primes[1..4];
var c = primes[..3];
var d = primes[1..];
```

A range expression `beg .. end` specifies a closed range of numbers or enum values. In a slice expression, the first element defaults to `0` and the last one defaults to `len - 1`. The inequalities `beg >= 0` and `end >= beg - 1` must hold when slicing. These rules allow using `a[0..-1]` to construct a zero length slice to the beginning of `a`.

The type `[][]T` is a multidimensional slice. The type `[]([]T)` is a slice of slices. A multidimensional slice points to a contiguous block of memory.

2.13 Bitwise Operations

The infix operators `>>`, `<<`, `^`, `~`, `&`, `|` are the usual bitwise operators. The infix operator `>>>` is the arithmetic shift right.

2.14 scope statements

`scope(exit)` statement allows statements to be executed on function exit.

`scope(success)` statement allows statements to be executed on successful function exit.

`scope(failure)` statement allows statements to be executed on error function exit.

Chapter 3

Functions

Functions are defined using **fn** keyword. The type of a function includes the types of parameters and its return type. All functions are known at compile-time and a variable cannot have **fn** type. But pointers to functions are allowed.

Nested functions are allowed and mainly acts as a namespacing mechanism. If the nested function accessess the outer function's local variables, then a pointer to the nested function cannot be returned from the function, cannot be stored into variables, and a pointer to it cannot be passed to other functions that do.

3.1 Tail-call Elimination

A **become** statement is the same as **return** except that it is guaranteed not to push an additional stack frame. This is only valid if the called function is explicitly named i.e., the tail-call is not through a function pointer.

3.2 Lambda Functions

Lambda functions are allowed in some contexts where there is no need for dynamic allocation to use the lambda. This is possible if the lambda doesn't access any variables from its environment or all calls to the lambda function can be inlined in the context its capturing. For example, when it is passed as a function pointer to a function that can be inlined and the called function only uses the function pointer to call it and not store it into other variables.

These rules also apply to nested functions.

```
fn map(x : []int,
       f : &fn(int) : int) : void {
    loop for (var e; x) {
        e = (*f)(e);
    }
```

```
}

fn foo() : void {
    var a = [1, 2, 3];
    var sum = 0;
    map(a[], \i => block { sum += i; i });
}

// compiled into
fn foo() : void {
    var a = [1, 2, 3];
    var sum = 0;
    loop for (var e; a[]) {
        e = block { sum += e; e };
    }
}
```

Chapter 4

Error Handling

The following are types that represent computations that can fail.

1. `*T` with `Null` as the error value.
2. **maybe**`!T` in module `std.`**maybe** with `Nothing` as the error value.
3. `result!(T, E)` in module `std.result` with `Err{e}` as the error value.
4. `errval!(T, val)` in module `std.err` where `val` is a literal of type `T` that is used to signal error. `T` must be an integral type.
5. `errvals!(T, vals)` in module `std.err` where `vals` represent a set of values of type `T` that is used to signal errors. `T` must be an integral type.
6. `errno!T` from the module `std.c.errno` signals errors through the global variable `errno_v`.
7. `errno_val!(T, val)` from the module `std.c.errno` signals errors by returning `val` and setting the global variable `errno_v` (For example, many POSIX system calls use this signalling mechanism). `T` must be an integral type.

Only the first three types should be used by modern code to signal errors. The other types are for interfacing conveniently with legacy `C` APIs. These types can only be used as return types of functions in `cweet`. When a function returning one of these types is called, it is automatically converted to a `result!(T, E)` or **maybe**`!T` in `cweet` code.

A function or expression that returns any of these types is considered as a function or expression that executes in an error context.

The following automatic conversions are allowed for ergonomical error handling.

For example, the `max` function below return `Null` if one of the inputs is `Null`.

```
fn max(x : *int, y : *int) : *int
{
    return if (*x > *y) x else y;
}
```

This function typechecks as follows:

1. The expressions `*x` and `*y` have type **maybe!**`int` and therefore the comparison expression does not typecheck.
2. The compiler unwraps all **maybe** from the expression. Now, this expression typechecks as `bool`. Since, it may fail, compiler lifts its type to **maybe!**`bool`.
3. Similarly, the **if else** has type **maybe!**`(*int)`.
4. A `Nothing` is mapped to `Null` to convert a **maybe!**`(*int)` to `*int`.

Chapter 5

Modules

```
// This file defines a module named 'file' and
// exports the following objects defined
// in this file.
module file export (
    File,
    open,
    read,
    write,
    close,
);

// export the type Point along with all fields.
// Also, export functions translate and isOrigin.
module geometry export (
    Point,
    translate, isOrigin
);

// import the module 'sys::mem'
// and use symbols 'alloc' and 'free' from it.
from sys::mem import (alloc, free,);

// use as mem::alloc, mem::free
from sys::mem import qualified mem (alloc, free);

// use mem::alloc and others. But, hide free
from sys::mem import qualified mem hiding (free);
```

Chapter 6

any and Polymorphism

The pointer types `&any` `*any` play the role of `void *` in `C`. The cast operator `any_cast!(*T)(ptr)` from the module `std::any` can be used to convert from or to such pointers. It is undefined behaviour to convert a `*any` to a pointer that was not the original type.

Chapter 7

`repr` and Type Aliasing

The `repr` type plays the role of `std::byte` in `C++`. Any object can be converted into a `[]repr` by using the function `repr` from `std::repr` module. This can be used to reinterpret the bits of one object as another.

A pointer to `repr` is the only pointer type that supports pointer arithmetic. The following operations are supported.

1. `(p : &repr + i : int) : &repr` points to `i` bytes after `p`.
2. `(p : &repr - q : &repr) : int`. The pointers `p` and `q` has to point to the same memory block. A pointer to one element past the allocated array is considered a pointer to the memory block.

Type aliased memory access is not undefined but implementation defined.

Chapter 8

Data Layout

Each type `T` has:

1. Size given by `@sizeof(:T)`
2. Alignment given by `@alignof(:T)`
3. Byte ordering that is `@le, @be, @me` where `@me` is the ordering used by the target architectures. This can be queried using `@endian(:T, @le)`.
4. Offset within the byte. This can be queried using `@bitoffset(:T)`. This is non-zero only for bitfields.

The data layout is unspecified unless the declaration specifies the layout. All fundamental data types follow layout for the corresponding `C` data types. The attribute `@repr(C)` forces **struct, union**, and tuples to follow `C` layout rules.

The attribute `@repr(packed)` can be used to declare packed structs. The programmer is given complete control of layout of data.

The byte ordering attributes apply to all the multibyte members of a packed structure. This can be overridden for individual members.

Here is the declaration of IEEE754 32-bit floating point format stored in a big-endian format with 32-bit alignment.

```
from std::bits import (bits);
type float = @repr(packed) @align(4) @be struct {
    sign     : bits!1,
    exponent : byte,
    mantissa : bits!23,
};
```

For example, the following declaration specifies the layout of an IPV4 packet.

```
from std::varray import (varray);

type IPV4_Packet = @repr(packed) @be struct {
    version : bits!4,
    ihl     : unsigned!4, // modulo 2^4.
    dscp    : bits!6,
```

```
    ecn     : bits!2,
    len     : u16,
    ...
    data    : varray!byte,
};
```

The attribute `@pad(n)` pads to the next `n` bit boundary. The attribute `@skip(n)` skips the next `n` bits. The corresponding data type has to be `void`.

A field in a `@repr(packed)` type can be:

1. `void` with `@pad` or `@skip`.
2. Any integral type including enums based on integral types.
3. `bool` Takes up a bit.
4. `unsigned!n` unsigned int of n bits.
5. `signed!n` signed int of n bits.
6. `bits!n` n bits with operations same as byte.
7. Arrays of allowed types.
8. struct or union with only allowed types.

The compiler emits the required instructions to handle the endianness, alignment, and packing within bytes. Notice that these types can only be used to match the format of data in memory obtained from an external source. The compiler generates instructions assuming that data resides in memory. These are not suitable for handling hardware-mapped memory which may have special requirements when reading from or writing to bits (such as volatile data).