CSE 220: Systems Programming

Aggregate Data Types

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo

Aggregate Types

C has two kinds of aggregate type: arrays and structures.

To define an aggregate type, it is helpful to first define a scalar type.

A scalar type is a type that contains a single value.

In C, scalar types are:

- arithmetic types (integers and floating point numbers)
- pointers (which we have learned are special integers)

Aggregate types are collections of scalar values.

Arrays

We have already seen arrays.

Arrays are a collection of values of the same type.

Arrays of the same type can differ in size.

```
int a1[10];      // integer array
int a2[3];       // integer array
```

The type stored in an array may be either scalar or aggregate.

Structures

Structures are a collection of values of mixed type.

Like arrays, structure types:

- May contain more than one value
- Can contain either scalar or aggregate types

Unlike arrays, structure types:

- Can contain values of the same type or different types
- Have instances that all have the same size
- Have named members

Unions

C has a third kind of related type, unions.

Unions are like structures in declaration, but unlike in usage.

Every member of a union occupies the same memory.

This means that writing to one member destroys the other members.

Unions are not a true aggregate type.

We will not discuss unions further (for now).

Memory Layout

Many data types must be located in memory according to certain rules.

In most cases, this is not obvious to the programmer.

Aggregate types, and pointers to aggregate types, expose this.

We will explore alignment and stride.

The C Struct

A `struct` is an aggregate data type consisting of one or more other types.

```
struct IntList {
    int              value;
    struct IntList *next;
};
```

This struct contains an integer and a pointer.

`value` and `next` are called members of the structure.

Any variable of type `struct IntList` contains both of these members.

Declaring and Using Structures

The syntax for structure declaration is

```
struct StructureTypeName {
    // Members in structure
    // Each member has a type and a name
} instance; // semicolon required!
```

An instance of the structure may be created where the structure is declared, or using the type name later:

```
struct StructureTypeName instance;
```

Accessing Structure Members

The . operator is used to access the members of a structure.

```
struct IntList node;
node.value = 3;
node.next = NULL;
```

Any member of a structure can be accessed with .:

```
struct ComplexList {
    struct Complex {
        double real, im;
    } complex;
    struct ComplexList *next;
} complexlist;
complexlist.complex.real = 0.0;
```

Structure Pointers

The `.` operator is cumbersome for structure pointers:

```
struct IntList *list = malloc(sizeof(struct IntList));
(*list).next = NULL;
```

The `->` operator is syntactic sugar for `(*).`:

```
list->next = NULL;
```

The `->` operator can be used to access any member of a structure via a pointer to the structure type.

Operations on Structures

A structure value:

- Can have its address taken with &
- Can be copied with =
- Can be used to access a member with .

A structure pointer:

- Can do all the things any pointer can do
- Can be used to access a member with ->

No other operations on structures are legal!

Structure Layout

The members of a structure are adjacent in memory.

This is similar to scalars in an array.

However, there are additional considerations regarding layout.

Recall that:

- The memory bus has a certain width
- Memory transfers data in words

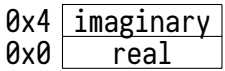
For this reason, we align data according to its type.

Simple Layout

In the simple case, members are adjacent.

Every member is laid out in order.

```
struct ComplexFloat {
  float real;
  float imaginary;
};
```



Alignment and Padding

In more complicated cases, alignment becomes an issue.

Data values are aligned so that multiple memory fetches are not required to retrieve a single value.

In general, this means that the address of a value must be evenly divisible by the size of its type.
Thus, if an `int` is 32 bits, its address is divisible by 4.
(32 bits / 8 bits per byte - 4 bytes, addressed in bytes)

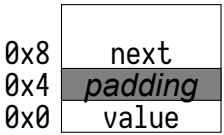
Padding is inserted between values to bring them into alignment.

Padding is unused memory and you cannot assume its value.

Struct Padding

In a structure, padding is applied between values.

```
struct IntList {
    int              value;
    struct IntList *next;
};
```



This struct is 16 bytes and contains 4 bytes of padding.

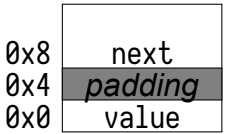
Struct Alignment

For padding in structures to work, the struct must be aligned.

Consider the previous example:

- If the address of the struct is divisible by 4, `value` is aligned, but `next` might not be
- If the address of the struct is divisible by 8, both are aligned

The struct itself is ordinarily aligned to the requirements of its largest member.



Alignment and Allocation

Recall that the standard allocator doesn't know what you're allocating.

For this reason, `malloc()` `et al.` normally align to the largest system requirement.

This ensures that any properly aligned structure will be aligned.

This leads to overhead which can cause significant waste.

We'll see much more about this later.

Stride

Stride is closely related to alignment, yet different.

Stride is the difference between two pointers to adjacent values of a particular type.

For simple types, stride is the same as size.

For example:

- If `int` is 32 bits, `sizeof(int)` is 4 and the stride of `int *` is 4.
- If `double` 64 bits, `sizeof(double)` is 8 and the stride of `double *` is 8.

For aggregate types, this can get more complicated.

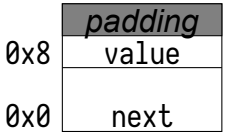
`void *` is a special case, and its stride is 1.

Stride in Aggregate Types

Consider this struct:

```
struct IntList {
    struct IntList *next;
    int             value;
};
```

It lays out in memory like this:



Padding here is to adjust stride to preserve alignment.

Pointer Arithmetic

Pointers are integer types, and can be computed.

Pointer arithmetic operates in stride-sized chunks.
(This is why pointers can dereference like arrays!)

```
double *dptr = &somedouble;
```

If the value of `dptr` were `0`, `dptr + 1` would be eight, not one!
This is because a double is 8 bytes wide.

Pointer Arithmetic — Aggregate Types

Stride for aggregate types can be quite large.

Consider:

```
struct Big {
    char array[256];
};
struct Big *b = NULL;
```

In this case, `b + 1` is the address 256!

Summary

- Integers, pointers, and floating point numbers are scalar types.
- Arrays and structures are aggregate types.
- Structures can contain members of mixed type.
- Scalar types must be aligned.
- Aggregate types must align for scalars.
- Allocation normally aligns to the largest type.
- Pointer arithmetic uses stride in computations.
- `void *` has a alertstride of 1.

Next Time …

- The compiler and toolchain

References I

Required Readings

[1] Randal E. Bryant and David R. O'Hallaron. *Computer Science: A Programmer's Perspective*. Third Edition. Chapter 8: 3.8.1, 3.8.2, 3.9.1, 3.9.3. Pearson, 2016.

[2] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Second Edition. Chapter 5: 5.3, 5.4, 5.7; Chapter 6: Intro, 6.1-6.7. Prentice Hall, 1988.

License

To retrieve a copy of this material, or related materials, see https://www.cse.buffalo.edu/~eblanton/.