C structures and unions

C structures: aggregate, yet scalar

- aggregate in that they hold multiple data items at one time
  - named *members* hold data items of various types
  - like the notion of class/field in C or C++
    – but without the data hiding features
- scalar in that C treats each structure as a unit
  - as opposed to the “array” approach: a pointer to a collection of members in memory
  - entire structures (not just pointers to structures) may be passed as function arguments, assigned to variables, etc.
  - Interestingly, they cannot be compared using ==
    (rationale: too inefficient)

Structure declarations

- Combined variable and type declaration

```
struct tag {member-list} variable-list;
```

- Any one of the three portions can be omitted

```
struct {int a, b; char *p;} x, y;  /* omit tag */
```

  - variables `x, y` declared with members as described: `int` members `a, b` and `char` pointer `p`.
  - `x` and `y` have same type, but differ from all others – even if there is another declaration:

```
struct {int a, b; char *p;} z;
/* z has different type from x, y */
```

Structure declarations

```
struct S {int a, b; char *p;};  /* omit variables */
```

- No variables are declared, but there is now a type `struct S` that can be referred to later

```
struct S z;  /* omit members */
```

  - Given an earlier declaration of `struct S`, this declares a variable of that type

```
typedef struct {int a, b; char *p;} S;
  /* omit both tag and variables */
```

  - This creates a simple type name `S` (more convenient than `struct S`)

Recursively defined structures

- Obviously, you can't have a structure that contains an instance of itself as a member – such a data item would be infinitely large
- But within a structure you can *refer* to structures of the same type, via pointers

```
struct TREENODE {
  char *label;
  struct TREENODE *leftchild, *rightchild;
}
```

Recursively defined structures

- When two structures refer to each other, one must be declared in incomplete (prototype) fashion

```
struct HUMAN;
struct PET {
  char name[NAME_LIMIT];
  char species[NAME_LIMIT];
  struct HUMAN *owner;
} fido = {″Fido″, ″Canis lupus familiaris″};
struct HUMAN {
  char name[NAME_LIMIT];
  struct PET pets[PET_LIMIT];
} sam = {″Sam″, {fido}};
```

We can't initialize the `owner` member at this point, since it hasn't been declared yet

Member access

- Direct access operator `s.m`
  - subscript and dot operators have same precedence and associate left-to-right, so we don't need parentheses for `sam.pets[0].species`
- Indirect access `s->m`: equivalent to `(*s).m`
  - Dereference a pointer to a structure, then return a member of that structure
  - Dot operator has higher precedence than indirection operator, so parentheses are needed in (*s).m

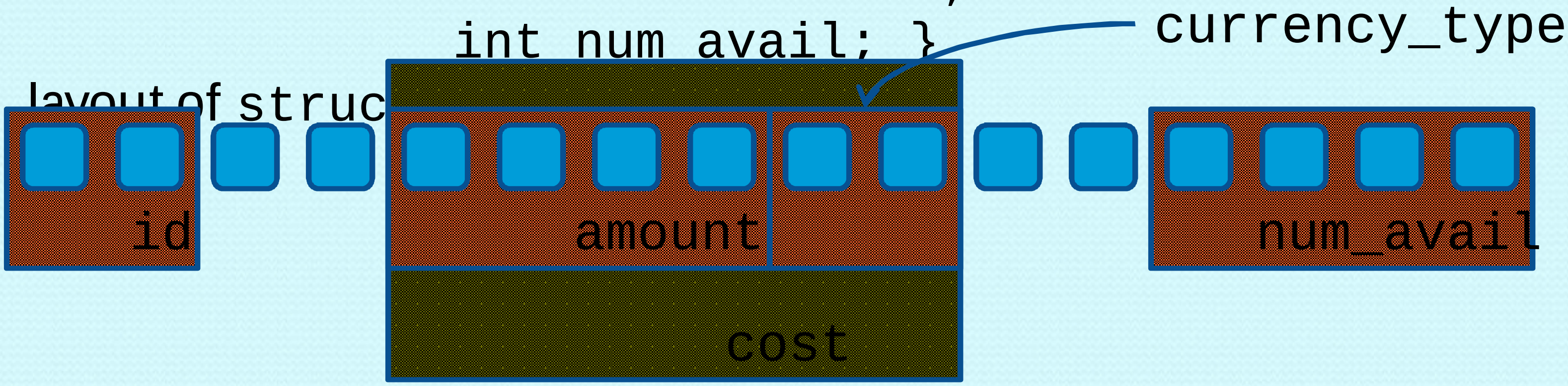
`.` evaluated first: access `owner` member
`*` evaluated next: dereference pointer to HUMAN

`.` and `->` have equal precedence and associate left-to-right

Memory layout

```
struct COST { int amount;
              char currency_type[2]; }
struct PART { char id[2];
              struct COST cost;
              int num_avail; }
```

layout of struc

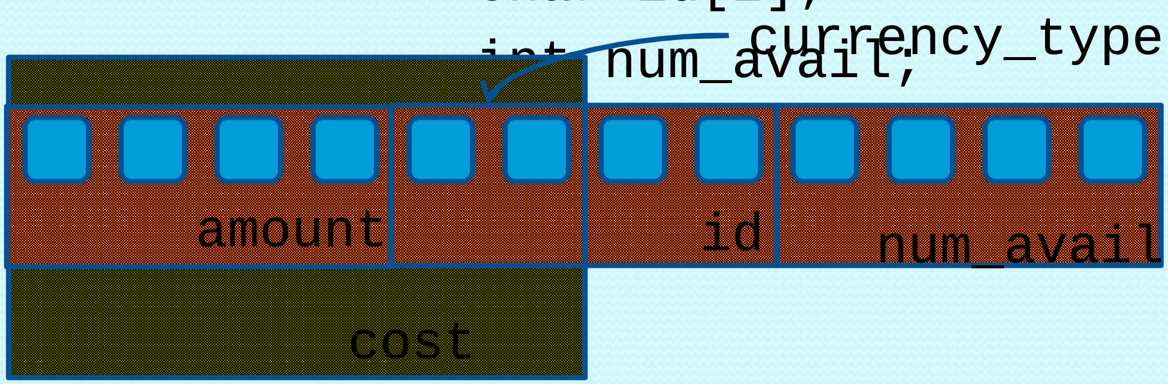


Here, the system uses 4-byte alignment of integers,
so `amount` and `num_avail` must be aligned
Four bytes wasted for each structure!

Memory layout

A better alternative (from a space perspective):

```
struct COST { int amount;
              char currency_type; }
struct PART { struct COST cost;
              char id[2];
              int num_avail;
```



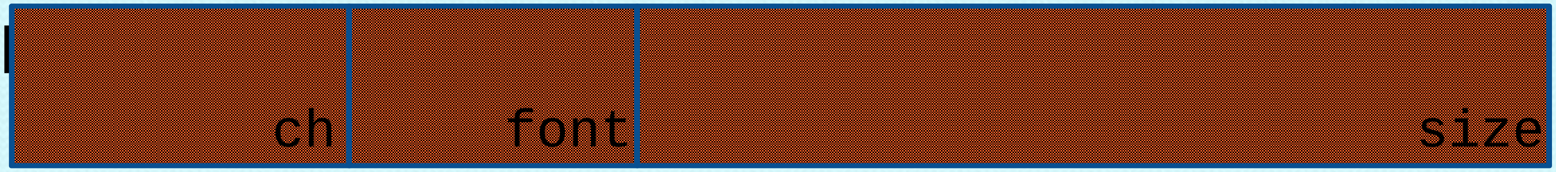
Bit fields

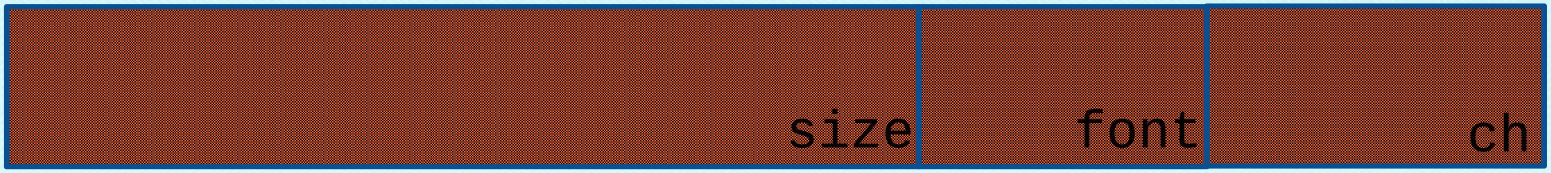
Bit field members must be `int`s

- If space is a serious concern, you can … of bits used for each member

Note: This won't work on machines with 16-bit `int`s

```
struct CHAR { unsigned ch: 7;
              unsigned font: 6;
              unsigned size: 19; };
```





Bit fields

- Portability is an issue:
  - Do any bit field sizes exceed the machine's `int` size?
  - Is there any pointer manipulation in your code that assumes a particular layout?
- Bit fields are "syntactic sugar" for more complex shifting/masking
  - e.g. to get `font` value, mask off the `ch` and `size` bits, then shift right by 19
  - This is what *actually happens* in the object code – bit fields just make it look simpler at the source level

Structures as function arguments

- Structures are scalars, so they can be returned and passed as arguments – just like `int`s, `char`s

```
struct BIG changestruct(struct BIG s);
```

  - Call by value: temporary copy of structure is created
  - Caution: passing large structures is inefficient
    – involves a lot of copying
- avoid by passing a pointer to the structure instead:

```
void changestruct(struct BIG *s);
```

- What if the `struct` argument is read-only?
  - Safe approach: use `const`

```
void changestruct(struct BIG const *s);
```

Unions

- Like structures, but every member occupies the same region of memory!
  - Structures: members are “and”ed together: “name and species and owner”
  - Unions: members are “xor”ed together

```
union VALUE {
  float f;
  int i;
  char *s;
};
/* either a float xor an int xor a string */
```

Unions

- Up to programmer to determine how to interpret a union (i.e. which member to access)
- Often used in conjunction with a “type” variable that indicates how to interpret the union value

```
enum TYPE { INT, FLOAT, STRING };
struct VARIABLE {
  enum TYPE type;
  union VALUE value;
};
```

Access `type` to determine how to interpret `value`

Unions

- Storage
  - size of union is the size of its largest member
  - avoid unions with widely varying member sizes;

    for the larger data types, consider using pointers instead
- Initialization
  - Union may only be initialized to a value appropriate for the type of its first member