C Structures &
Dynamic Memory Management

Goals of this Lecture

- Help you learn about:
  - Structures and unions
  - Dynamic memory management
- Note:
  - Will be covered in precepts as well
  - We look at them in more detail

Structure Variables

- Structure: collection of related data items
- Comparison with array
  - The elements of a structure (its ***members***) aren't required to have the same type.
  - The members of a structure have names; to select a particular member, we specify its name, not its position.
- Structures are often called ***records,*** and members are known as ***fields.***

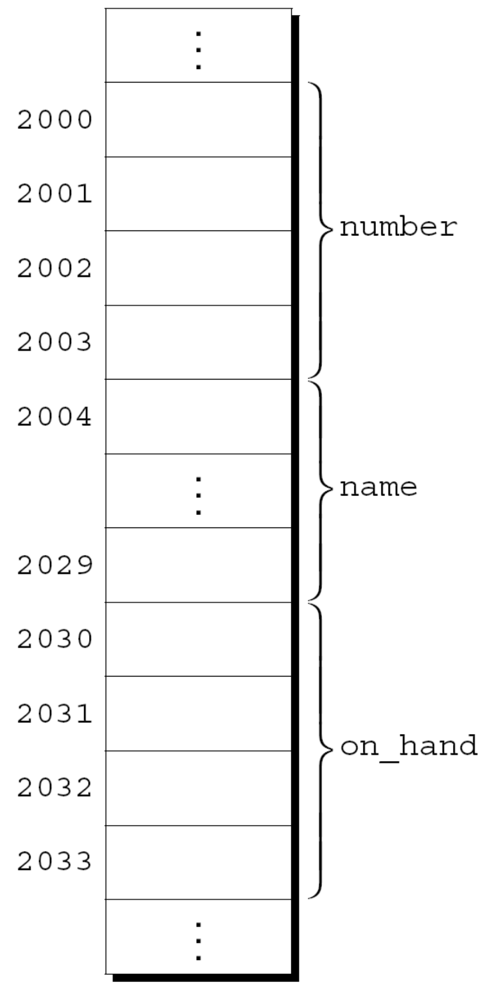
Declaring Structure Variables

- A declaration of two structure variables that store information about parts in a warehouse:

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```

Declaring Structure Variables

- The members of a structure are stored in memory in the order in which they're declared.
- Appearance of `part1`
- Assumptions:
  - `part1` is located at address 2000.
  - Integers occupy four bytes.
  - `NAME_LEN` has the value 25.
  - There are no gaps between the members.

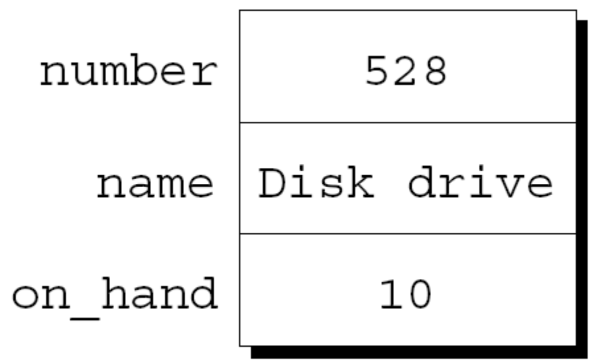


Initializing Structure Variables

- A structure declaration may include an initializer:

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1 = {528, "Disk drive", 10},
  part2 = {914, "Printer cable", 5};
```

- Appearance of `part1` after initialization:



Initializing Structure Variables

- Structure initializers follow rules similar to those for array initializers.
- Expressions used in a structure initializer must be constant. (relaxed in C99)
- An initializer can have fewer members than the structure it's initializing.
- Any "leftover" members are given 0 as their initial value.

Designated Initializers (C99)

- The initializer for `part1` shown in the previous example:

```
{528, "Disk drive", 10}
```

- In a designated initializer, each value would be labeled by the name of the member that it initializes:

```
{.number = 528, .name = "Disk drive", .on_hand = 10}
```

- The combination of the period and the member name is called a ***designator***.

Designated Initializers (C99)

- Not all values listed in a designated initializer need be prefixed by a designator.
- Example:

```
{.number = 528, "Disk drive", .on_hand = 10}
```

  The compiler assumes that `"Disk drive"` initializes the member that follows `number` in the structure.
- Any members that the initializer fails to account for are set to zero.

Operations on Structures

- Accessing a member within a structure:

```
name.member
```

- Statements that display the values of `part1`'s members:

```
printf("Part number: %d\n", part1.number);
printf("Part name: %s\n", part1.name);
printf("Quantity on hand: %d\n", part1.on_hand);
```

Operations on Structures

- The members of a structure are lvalues.
- They can appear on the left side of an assignment or as the operand in an increment or decrement expression:

```
part1.number = 258;
 /* changes part1's part number */
part1.on_hand++;
 /* increments part1's quantity on hand */
```

Operations on Structures

- The **period** used to access a structure member is actually a *C* **operator**.
- It takes precedence over nearly all other operators.
- Example:

```
scanf("%d", &part1.on_hand);
```

The `.` operator takes precedence over the `&` operator, so `&` computes the address of `part1.on_hand`.

Operations on Structures

- The other major structure operation is assignment:

```
part2 = part1;
```

- The effect of this statement is to copy all members from part1 to part2.
  - `part1.number` into `part2.number`, `part1.name` into `part2.name`, and so on.

Operations on Structures

- Arrays can't be copied using the = operator, but an **array embedded within a structure is copied** when the enclosing structure is copied.
- Some programmers exploit this property by creating "dummy" structures to enclose arrays that will be copied later:

```
struct { int a[10]; } a1, a2;
a1 = a2;
/* legal, since a1 and a2 are structures
   a1.a[i] = a2.a[i];  (0 <= i <= 9)  */
```

Operations on Structures

- The = operator can be used only with structures of ***compatible*** types.
  - Two structures declared at the same time (as `part1` and `part2` were) are compatible.
  - Structures declared using the same “structure tag” or the same type name are also compatible.

- Other than assignment, C provides no operations on entire structures.
  - **In particular, the == and != operators can’t be used with structures.**

Structure Types

- Suppose that a program needs to declare several structure variables with identical members.
- Ways to name a structure:
  - Declare a "structure tag"
  - Use `typedef` to define a type name

Declaring a Structure Tag

- A ***structure tag*** is a name used to identify a particular kind of structure.
- The declaration of a structure tag named `part`:

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
};
```

- Note that a semicolon must follow the right brace.

Declaring a Structure Tag

- The `part` tag can be used to declare variables:

  ```
  struct part part1, part2, *p;
  ```

  `p` can point to a struct part variable.

  ```
  p = &part1;
  ```

  `(*p).name` or `p->name` to access `part1.name`

- We can't drop the word `struct`:

  ```
  part part1, part2;   /*** WRONG ***/
  ```

  `part` isn't a type name; without the word `struct`, it is meaningless.

- Since structure tags aren't recognized unless preceded by the word `struct`, they don't conflict with other names used in a program.

Declaring a Structure Tag

- The declaration of a structure *tag* can be combined with the declaration of structure *variables:*

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```

Declaring a Structure Tag

- All structures declared to have type `struct part` are compatible with one another:

```
struct part part1 = {528, "Disk drive", 10};
struct part part2;

part2 = part1;
  /* legal; both parts have the same type */
```

Defining a Structure Type

- As an alternative to declaring a structure tag, we can use **`typedef`** to define a genuine type name.

- A definition of a type named `Part`:

```
typedef struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} Part;
```

- `Part` can be used in the same way as built-in types:

```
Part part1, part2;
```

Defining a Structure Type

- When it comes time to name a structure, we can usually choose either to declare a structure tag or to use `typedef`.

- However, declaring a structure tag is mandatory when the structure itself is referenced in it

```
typedef struct tagList {
    char *key;
    int value;
    struct tagList *next;
    } List;
```

Nested Arrays and Structures

- Structures and arrays can be combined without restriction.
- Arrays may have structures as their elements, and structures may contain arrays and structures as members.

Nested Structures

- Suppose that `person_name` is the following structure:

```
struct person_name {
  char first[FIRST_NAME_LEN+1];
  char middle_initial;
  char last[LAST_NAME_LEN+1];
};
```

- We can use `person_name` as part of a larger structure:

```
struct student {
  struct person_name name;
  int id, age;
  char sex;
} student1, student2;
```

- Accessing `student1`'s first name, middle initial, or last name requires two applications of the . operator:

```
strcpy(student1.name.first, "Fred");
```

Arrays of Structures

- An array of `part` structures capable of storing information about 100 parts:

```
struct part inventory[100];
```

- Accessing a member within a `part` structure requires a combination of subscripting and member selection:

```
inventory[i].number = 883;
```

- Accessing a single character in a part name requires subscripting, followed by selection, followed by subscripting:

```
inventory[i].name[0] = '\0';
```

Initializing an Array of Structures

- One reason for initializing an array of structures is that it contains information that won't change during program execution.
- Example: an array that contains country codes used when making international telephone calls.
- The elements of the array will be structures that store the name of a country along with its code:

```
struct dialing_code {
  char *country;
  int code;
};
```

Initializing an Array of Structures

```
const struct dialing_code country_codes[] =
  {{"Argentina",            54}, {"Bangladesh",      880},
   {"Brazil",               55}, {"Burma (Myanmar)",  95},
   {"China",                86}, {"Colombia",         57},
   {"Congo, Dem. Rep. of", 243}, {"Egypt",            20},
   {"Ethiopia",            251}, {"France",           33},
   {"Germany",              49}, {"India",            91},
   {"Indonesia",            62}, {"Iran",             98},
   {"Italy",                39}, {"Japan",            81},
   {"Mexico",               52}, {"Nigeria",         234},
   {"Pakistan",             92}, {"Philippines",      63},
   {"Poland",               48}, {"Russia",            7},
   {"South Africa",         27}, {"South Korea",      82},
   {"Spain",                34}, {"Sudan",           249},
   {"Thailand",             66}, {"Turkey",           90},
   {"Ukraine",             380}, {"United Kingdom",   44},
   {"United States",         1}, {"Vietnam",          84}};
```

- The inner braces around each structure value are optional.

Unions

- *A **union**,* like a structure, consists of one or more members, possibly of different types.
- The compiler allocates only enough space for **the largest of the members**, which overlay each other within this space.
- Assigning a new value to one member alters the values of the other members as well.

Unions

- An example of a union variable:

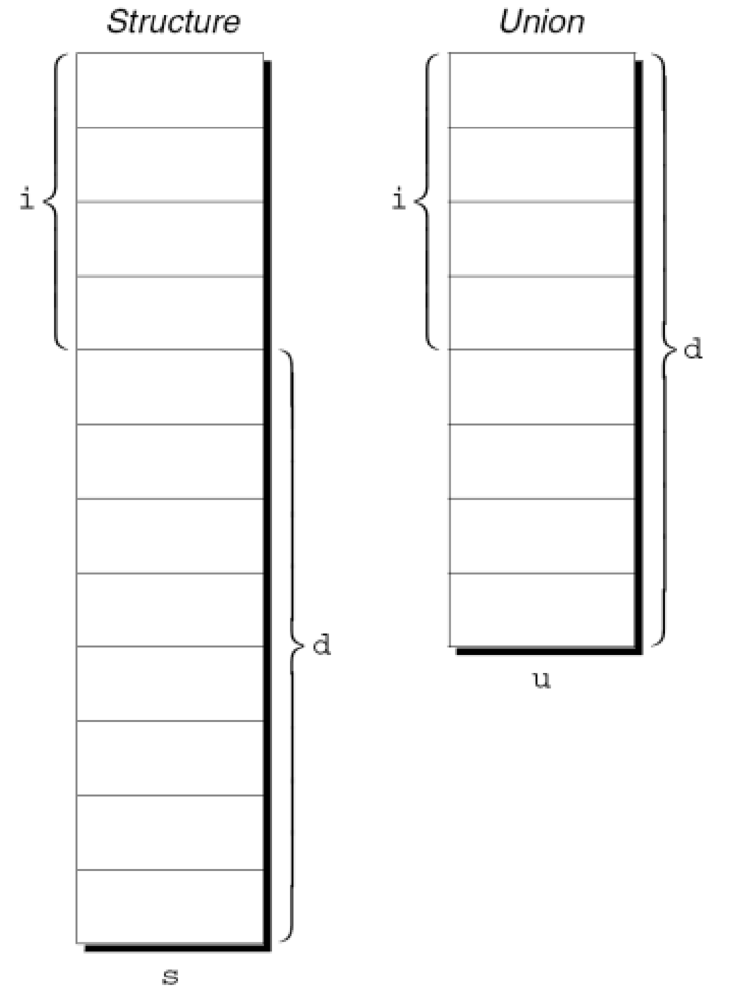
```
union {
  int i;
  double d;
} u;
```

- The declaration of a union closely resembles a structure declaration:

```
struct {
  int i;
  double d;
} s;
```

Unions

- The structure `s` and the union `u` differ in just one way.
- The members of `s` are stored at different addresses in memory.
- The members of `u` are stored at the same address.



Unions

- Members of a union are accessed in the same way as members of a structure:

```
u.i = 82;
u.d = 74.8;
```

- Changing one member of a union alters any value previously stored in any of the other members.
  - Storing a value in `u.d` causes any value previously stored in `u.i` to be lost.
  - Changing `u.i` corrupts `u.d`.

Unions

- The properties of unions are almost identical to the properties of structures.
- We can declare union tags and union types in the same way we declare structure tags and types.
- Like structures, unions can be copied using the = operator, passed to functions, and returned by functions.

Unions

- Only the first member of a union can be given an initial value.
- How to initialize the `i` member of `u` to 0:

```
union {
  int i;
  double d;
} u = {0};
```

- The expression inside the braces must be constant. (The rules are slightly different in C99.)

Unions

- Designated initializers can also be used with unions.
- A designated initializer allows us to specify which member of a union should be initialized:

```
union {
  int i;
  double d;
} u = {.d = 10.0};
```

- Only one member can be initialized, but it doesn't have to be the first one.

Unions

- Applications for unions:
  - Saving space
  - Building mixed data structures
  - See King's book.

Dynamic Storage Allocation

- C's data structures, including arrays, are normally fixed in size.

- Fixed-size data structures can be a problem, since we're forced to choose their sizes when writing a program.

- Fortunately, C supports ***dynamic storage allocation:*** the ability to allocate storage during program execution.

- Using dynamic storage allocation, we can design data structures that grow (and shrink) as needed.

Memory Allocation Functions

- The `<stdlib.h>` header declares three memory allocation functions:

  `malloc`—Allocates a block of memory but doesn't initialize it.

  `calloc`—Allocates a block of memory and clears it.

  `realloc`—Resizes a previously allocated block of memory.

- These functions return a value of type `void *` (a "generic" pointer).
  - If a memory allocation function can't locate a memory block of the requested size, it returns a ***null pointer.*** (NULL or 0)

Null Pointers

- An example of testing `malloc`'s return value:

```
p = malloc(10000);
if (p == NULL) {
/* allocation failed; take appropriate action */
}
```

- `NULL` is a macro (defined in various library headers) that represents the null pointer.

- Some programmers combine the call of `malloc` with the `NULL` test:

```
if ((p = malloc(10000)) == NULL) {
/* allocation failed; take appropriate action */
}
```

Using `malloc` to Allocate Memory

- Prototype for the `malloc` function:

```
void *malloc(size_t size);
```

- `malloc` allocates a block of `size` bytes and returns a pointer to it.
- `size_t` is an unsigned integer type defined in the library.

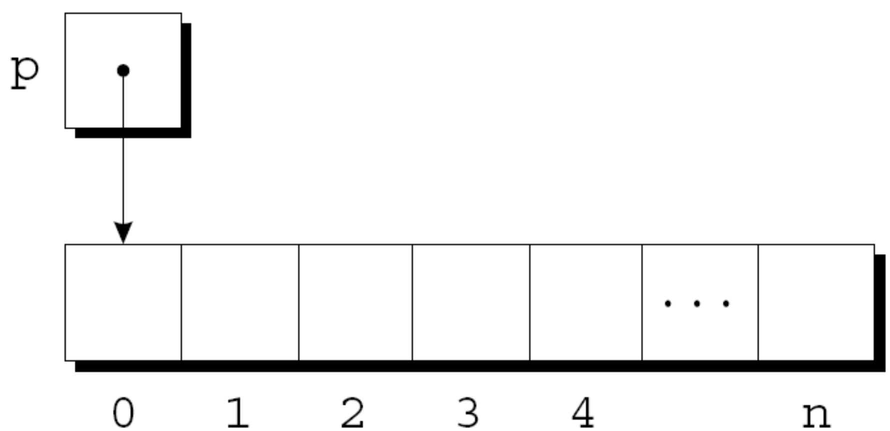
Using **`malloc`** to Allocate Memory for a String

- A call of `malloc` that allocates memory for a string of `n` characters:

  ```
  p = (char *)malloc(n + 1);
  ```

  `p` is a `char *` variable.

- Each character requires one byte of memory; adding 1 to `n` leaves room for the null character.

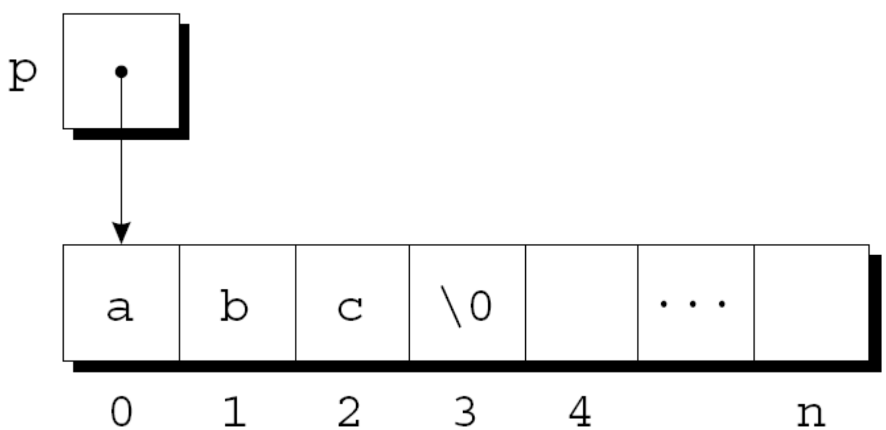


Using **`malloc`** to Allocate Memory for a String

- Calling `strcpy` is one way to initialize this array:

```
strcpy(p, "abc");
```

- The first four characters in the array will now be `a`, `b`, `c`, and `\0`:



Using **`malloc`** to Allocate Storage for an Array

- Suppose a program needs an array of `n` integers, where `n` is computed during program execution.
- We'll first declare a pointer variable:

```
int *a;
```

- Once the value of `n` is known, the program can call `malloc` to allocate space for the array:

```
a = malloc(n * sizeof(int));
```

- Always use the `sizeof` operator to calculate the amount of space required for each element.

Using **`malloc`** to Allocate Storage for an Array

- We can now ignore the fact that `a` is a pointer and use it instead as an array name, thanks to the relationship between arrays and pointers.

- For example, we could use the following loop to initialize the array that `a` points to:

```
for (i = 0; i < n; i++)
  a[i] = 0;
```

- We also have the option of using pointer arithmetic instead of subscripting to access the elements of the array.

The **`calloc`** Function

- Prototype for `calloc`:

```
void *calloc(size_t nmemb, size_t size);
```

- Properties of `calloc`:
  - Allocates space for an array with `nmemb` elements, each of which is `size` bytes long.
  - Returns a null pointer if the requested space isn't available.
  - Initializes allocated memory by setting all bits to 0.

The **`calloc`** Function

- A call of `calloc` that allocates space for an array of `n` integers:

```
a = calloc(n, sizeof(int));
```

- By calling `calloc` with 1 as its first argument, we can allocate space for a data item of any type:

```
struct point { int x, y; } *p;

p = calloc(1, sizeof(struct point));
```

The **`realloc`** Function

- The `realloc` function can resize a dynamically allocated array.
- Prototype for `realloc`:

```
void *realloc(void *ptr, size_t size);
```

- `ptr` must point to a memory block obtained by a previous call of `malloc`, `calloc`, or `realloc`.
- `size` represents the new size of the block, which may be larger or smaller than the original size.

The **`realloc`** Function

- Properties of `realloc`:
  - When it expands a memory block, `realloc` doesn't initialize the bytes that are added to the block.
  - If `realloc` can't enlarge the memory block as requested, it returns a null pointer; the data in the old memory block is unchanged.
  - If `realloc` is called with a null pointer as its first argument, it behaves like `malloc`.
  - If `realloc` is called with 0 as its second argument, it frees the memory block.

The `realloc` Function

- We expect `realloc` to be reasonably efficient:
  - When asked to reduce the size of a memory block, `realloc` should shrink the block "in place."
  - `realloc` should always attempt to expand a memory block without moving it.
- If it can't enlarge a block, `realloc` will allocate a new block elsewhere, then copy the contents of the old block into the new one.
- Once `realloc` has returned, be sure to update all pointers to the memory block in case it has been moved.

Deallocating Storage

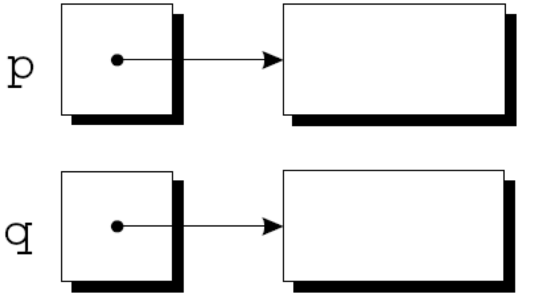
- `malloc` and the other memory allocation functions obtain memory blocks from a storage pool known as the ***heap***.
- Calling these functions too often—or asking them for large blocks of memory—can exhaust the heap, causing the functions to return a null pointer.
- To make matters worse, a program may allocate blocks of memory and then lose track of them, thereby wasting space.

Deallocating Storage

- Example:

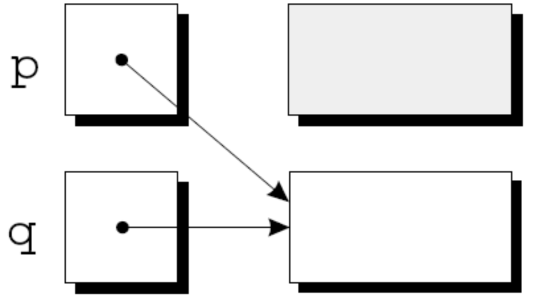
```
p = malloc(…);
q = malloc(…);
p = q;
```

- A snapshot after the first two statements have been executed:



Deallocating Storage

- After `q` is assigned to `p`, both variables now point to the second memory block:



- There are no pointers to the first block, so we'll never be able to use it again.

Deallocating Storage

- A block of memory that's no longer accessible to a program is said to be ***garbage.***
- A program that leaves garbage behind has a ***memory leak.***
- Some languages provide a ***garbage collector*** that automatically locates and recycles garbage, but C doesn't.
- Instead, each C program is responsible for recycling its own garbage by calling the `free` function to release unneeded memory.

The **`free`** Function

- Prototype for `free`:

```
void free(void *ptr);
```

- `free` will be passed a pointer to an unneeded memory block:

```
p = malloc(…);
q = malloc(…);
free(p);
p = q;
```

- Calling `free` releases the block of memory that `p` points to.

The “Dangling Pointer” Problem

- Using `free` leads to a new problem: ***dangling pointers.***
- `free(p)` deallocates the memory block that `p` points to, but doesn't change `p` itself.
- If we forget that `p` no longer points to a valid memory block, chaos may ensue:

```
char *p = malloc(4);
…
free(p);
…
strcpy(p, "abc");   /*** WRONG ***/
```

- Modifying the memory that `p` points to is a serious error.

The “Dangling Pointer” Problem

- Dangling pointers can be hard to spot, since several pointers may point to the same block of memory.
- When the block is freed, all the pointers are left dangling.

Summary

- Structures and Unions
  - Allows heterogeneous data items
  - Structure tag or typedef can be used for specifying the same struct variables
- Dynamic memory management
  - Allocates variable-sized space on run-time
  - De-allocation is the programmer's responsibility: be careful about dangling pointers