Chapter 16

Structures, Unions, and Enumerations

Structure Variables

- The properties of a ***structure*** are different from those of an 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.
- In some languages, structures are called ***records,*** and members are known as ***fields.***

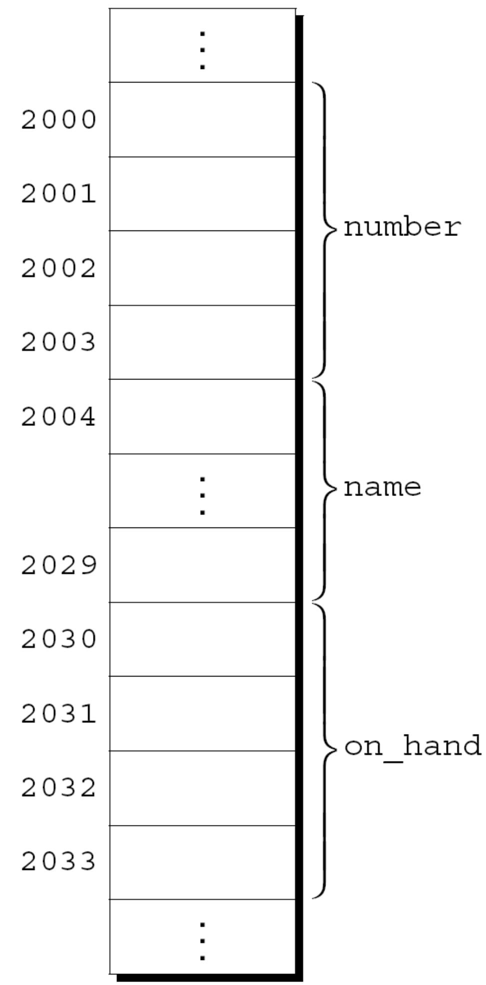
Declaring Structure Variables

- A structure is a logical choice for storing a collection of related data items.
- 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;
```

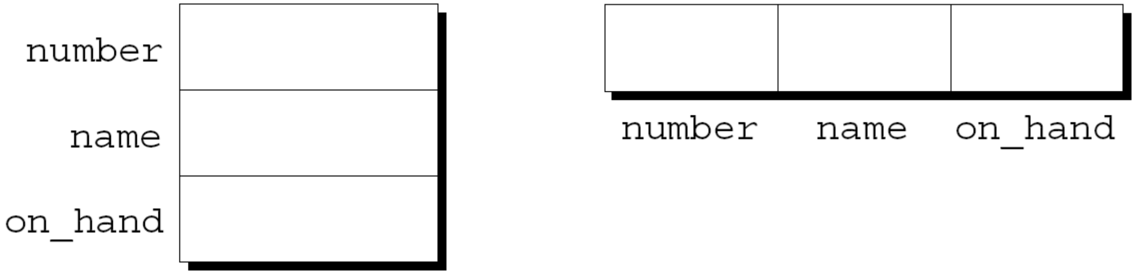
Declaring Structure Variables

- 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.



Declaring Structure Variables

- Abstract representations of a structure:



- Member values will go in the boxes later.

Declaring Structure Variables

- Each structure represents a new scope.
- Any names declared in that scope won't conflict with other names in a program.
- In C terminology, each structure has a separate ***name space*** for its members.

Declaring Structure Variables

- For example, the following declarations can appear in the same program:

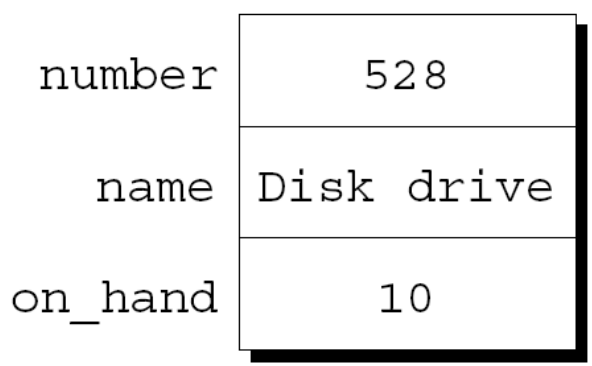
```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;

struct {
  char name[NAME_LEN+1];
  int number;
  char sex;
} employee1, employee2;
```

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. (This restriction is 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)

- C99's designated initializers can be used with structures.
- 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)

- Designated initializers are easier to read and check for correctness.
- Also, values in a designated initializer don’t have to be placed in the same order that the members are listed in the structure.

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

- To access a member within a structure, we write the name of the structure first, then a period, then the name of the 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 `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 */
```

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.
- We need a name that represents a *type* of structure, not a particular structure *variable*.
- 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;
```

- 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

- 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 the built-in types:

```
Part part1, part2;
```

Structures as Arguments and Return Values

- Functions may have structures as arguments and return values.
- A function with a structure argument:

```
void print_part(struct part p)
{
  printf("Part number: %d\n", p.number);
  printf("Part name: %s\n", p.name);
  printf("Quantity on hand: %d\n", p.on_hand);
}
```

- A call of `print_part`:

```
print_part(part1);
```

Structures as Arguments and Return Values

- A function that returns a `part` structure:

```
struct part build_part(int number,
                       const char *name,
                       int on_hand)
{
  struct part p;

  p.number = number;
  strcpy(p.name, name);
  p.on_hand = on_hand;
  return p;
}
```

- A call of `build_part`:

```
part1 = build_part(528, "Disk drive", 10);
```

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

- Nesting one structure inside another is often useful.
- 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];
};
```

Nested Structures

- 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");
```

Nested Structures

- Having `name` be a structure makes it easier to treat names as units of data.
- A function that displays a name could be passed one `person_name` argument instead of three arguments:

```
display_name(student1.name);
```

Arrays of Structures

- One of the most common combinations of arrays and structures is an array whose elements are structures.
- This kind of array can serve as a simple database.
- An array of `part` structures capable of storing information about 100 parts:

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

Arrays of Structures

- Accessing a `part` in the array is done by using subscripting:

```
print_part(inventory[i]);
```

- 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 member selection, followed by subscripting:

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

Initializing an Array of Structures

- Initializing an array of structures is done in much the same way as initializing a multidimensional array.
- Each structure has its own brace-enclosed initializer; the array initializer wraps another set of braces around the structure initializers.

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.

Program: Maintaining a Parts Database

- The `inventory.c` program illustrates how nested arrays and structures are used in practice.
- The program tracks parts stored in a warehouse.
- Information about the parts is stored in an array of structures.
- Contents of each structure:
  - Part number
  - Name
  - Quantity

Program: Maintaining a Parts Database

- Operations supported by the program:
  - Add a new part number, part name, and initial quantity on hand
  - Given a part number, print the name of the part and the current quantity on hand
  - Given a part number, change the quantity on hand
  - Print a table showing all information in the database
  - Terminate program execution

Program: Maintaining a Parts Database

- The codes `i` (insert), `s` (search), `u` (update), `p` (print), and `q` (quit) will be used to represent these operations.
- A session with the program:

```
Enter operation code: i
Enter part number: 528
Enter part name: Disk drive
Enter quantity on hand: 10

Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 10
```

Program: Maintaining a Parts Database

```
Enter operation code: s
Enter part number: 914
Part not found.

Enter operation code: i
Enter part number: 914
Enter part name: Printer cable
Enter quantity on hand: 5

Enter operation code: u
Enter part number: 528
Enter change in quantity on hand: -2
```

Program: Maintaining a Parts Database

```
Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 8

Enter operation code: p
Part Number   Part Name                 Quantity on Hand
    528       Disk drive                        8
    914       Printer cable                     5

Enter operation code: q
```

Program: Maintaining a Parts Database

- The program will store information about each part in a structure.
- The structures will be stored in an array named `inventory`.
- A variable named `num_parts` will keep track of the number of parts currently stored in the array.

Program: Maintaining a Parts Database

- An outline of the program's main loop:

```
for (;;) {
  prompt user to enter operation code;
  read code;
  switch (code) {
    case 'i': perform insert operation; break;
    case 's': perform search operation; break;
    case 'u': perform update operation; break;
    case 'p': perform print operation; break;
    case 'q': terminate program;
    default:  print error message;
  }
}
```

Program: Maintaining a Parts Database

- Separate functions will perform the insert, search, update, and print operations.
- Since the functions will all need access to `inventory` and `num_parts`, these variables will be external.
- The program is split into three files:
  - `inventory.c` (the bulk of the program)
  - `readline.h` (contains the prototype for the `read_line` function)
  - `readline.c` (contains the definition of `read_line`)

inventory.c

```
/* Maintains a parts database (array version) */

#include <stdio.h>
#include "readline.h"

#define NAME_LEN 25
#define MAX_PARTS 100

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

int num_parts = 0;   /* number of parts currently stored */

int find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);
```

```
/**********************************************************
 * main: Prompts the user to enter an operation code,    *
 *       then calls a function to perform the requested  *
 *       action. Repeats until the user enters the       *
 *       command 'q'. Prints an error message if the user *
 *       enters an illegal code.                         *
 **********************************************************/
int main(void)
{
  char code;
  for (;;) {
    printf("Enter operation code: ");
    scanf(" %c", &code);
    while (getchar() != '\n')   /* skips to end of line */
      ;
```

```
    switch (code) {
      case 'i': insert();
                break;
      case 's': search();
                break;
      case 'u': update();
                break;
      case 'p': print();
                break;
      case 'q': return 0;
      default:  printf("Illegal code\n");
    }
    printf("\n");
  }
}
```

```
/**********************************************************
 * find_part: Looks up a part number in the inventory     *
 *            array. Returns the array index if the part  *
 *            number is found; otherwise, returns -1.     *
 **********************************************************/
int find_part(int number)
{
  int i;

  for (i = 0; i < num_parts; i++)
    if (inventory[i].number == number)
      return i;
  return -1;
}
```

```
/**********************************************************
 * insert: Prompts the user for information about a new   *
 *         part and then inserts the part into the        *
 *         database. Prints an error message and returns  *
 *         prematurely if the part already exists or the  *
 *         database is full.                              *
 **********************************************************/
void insert(void)
{
  int part_number;

  if (num_parts == MAX_PARTS) {
    printf("Database is full; can't add more parts.\n");
    return;
  }
```

```
  printf("Enter part number: ");
  scanf("%d", &part_number);
  if (find_part(part_number) >= 0) {
    printf("Part already exists.\n");
    return;
  }

  inventory[num_parts].number = part_number;
  printf("Enter part name: ");
  read_line(inventory[num_parts].name, NAME_LEN);
  printf("Enter quantity on hand: ");
  scanf("%d", &inventory[num_parts].on_hand);
  num_parts++;
}
```

```
/**********************************************************
 * search: Prompts the user to enter a part number, then  *
 *         looks up the part in the database. If the part *
 *         exists, prints the name and quantity on hand;  *
 *         if not, prints an error message.               *
 **********************************************************/
void search(void)
{
  int i, number;

  printf("Enter part number: ");
  scanf("%d", &number);
  i = find_part(number);
  if (i >= 0) {
    printf("Part name: %s\n", inventory[i].name);
    printf("Quantity on hand: %d\n", inventory[i].on_hand);
  } else
    printf("Part not found.\n");
}
```

```
/**********************************************************
 * update: Prompts the user to enter a part number.      *
 *         Prints an error message if the part doesn't   *
 *         exist; otherwise, prompts the user to enter   *
 *         change in quantity on hand and updates the    *
 *         database.                                      *
 **********************************************************/
void update(void)
{
  int i, number, change;

  printf("Enter part number: ");
  scanf("%d", &number);
  i = find_part(number);
  if (i >= 0) {
    printf("Enter change in quantity on hand: ");
    scanf("%d", &change);
    inventory[i].on_hand += change;
  } else
    printf("Part not found.\n");
}
```

```
/**********************************************************
 * print: Prints a listing of all parts in the database,  *
 *        showing the part number, part name, and         *
 *        quantity on hand. Parts are printed in the      *
 *        order in which they were entered into the       *
 *        database.                                       *
 **********************************************************/
void print(void)
{
  int i;

  printf("Part Number   Part Name                  "
         "Quantity on Hand\n");
  for (i = 0; i < num_parts; i++)
    printf("%7d       %-25s%11d\n", inventory[i].number,
           inventory[i].name, inventory[i].on_hand);
}
```

readline.h

```
#ifndef READLINE_H
#define READLINE_H

/**********************************************************
 * read_line: Skips leading white-space characters, then  *
 *            reads the remainder of the input line and   *
 *            stores it in str. Truncates the line if its *
 *            length exceeds n. Returns the number of     *
 *            characters stored.                          *
 **********************************************************/
int read_line(char str[], int n);

#endif
```

readline.c

```
#include <ctype.h>
#include <stdio.h>
#include "readline.h"

int read_line(char str[], int n)
{
  int ch, i = 0;

  while (isspace(ch = getchar()))
    ;
  while (ch != '\n' && ch != EOF) {
    if (i < n)
      str[i++] = ch;
    ch = getchar();
  }
  str[i] = '\0';
  return i;
}
```

Chapter 17

Advanced Uses of Pointers

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.

Dynamic Storage Allocation

- Dynamic storage allocation is used most often for strings, arrays, and structures.
- Dynamically allocated structures can be linked together to form lists, trees, and other data structures.
- Dynamic storage allocation is done by calling a memory allocation function.

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).

Null Pointers

- If a memory allocation function can't locate a memory block of the requested size, it returns a ***null pointer*.**
- After we've stored the function's return value in a pointer variable, we must test to see if it's a null pointer.

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 */
}
```

Dynamically Allocated Strings

- Dynamic storage allocation is often useful for working with strings.
- Strings are stored in character arrays, and it can be hard to anticipate how long these arrays need to be.
- By allocating strings dynamically, we can postpone the decision until the program is running.

Using `malloc` to Allocate Memory for a String

- 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:

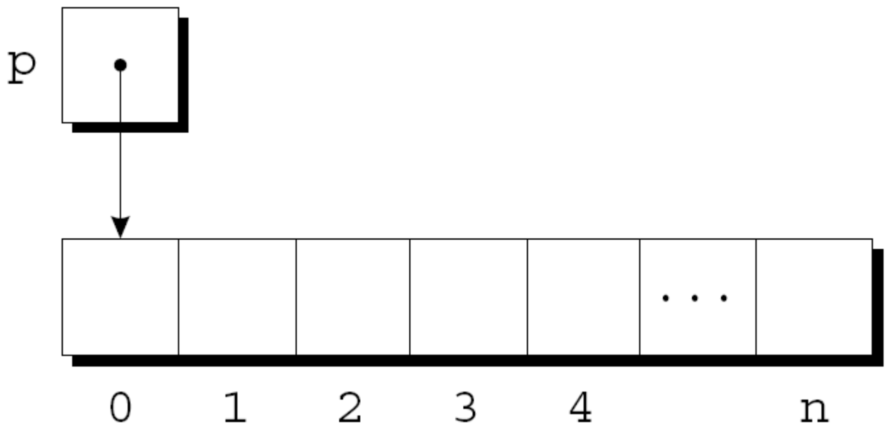
  ```
  p = malloc(n + 1);
  ```

  `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

- Memory allocated using `malloc` isn't cleared, so `p` will point to an uninitialized array of `n` + 1 characters:

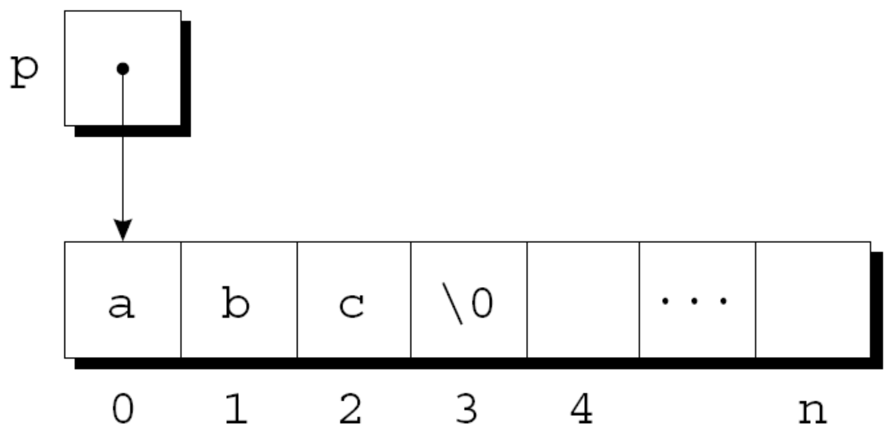


Using `malloc` to Allocate Memory for a String

- 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 Dynamic Storage Allocation in String Functions

- Dynamic storage allocation makes it possible to write functions that return a pointer to a "new" string.
- Consider the problem of writing a function that concatenates two strings without changing either one.
- The function will measure the lengths of the two strings to be concatenated, then call `malloc` to allocate the right amount of space for the result.

Using Dynamic Storage Allocation in String Functions

```
char *concat(const char *s1, const char *s2)
{
  char *result;

  result = malloc(strlen(s1) + strlen(s2) + 1);
  if (result == NULL) {
    printf("Error: malloc failed in concat\n");
    exit(EXIT_FAILURE);
  }
  strcpy(result, s1);
  strcat(result, s2);
  return result;
}
```

Using Dynamic Storage Allocation in String Functions

- A call of the `concat` function:

```
p = concat("abc", "def");
```

- After the call, `p` will point to the string `"abcdef"`, which is stored in a dynamically allocated array.

Using Dynamic Storage Allocation in String Functions

- Functions such as `concat` that dynamically allocate storage must be used with care.
- When the string that `concat` returns is no longer needed, we'll want to call the `free` function to release the space that the string occupies.
- If we don't, the program may eventually run out of memory.

Dynamically Allocated Arrays

- Dynamically allocated arrays have the same advantages as dynamically allocated strings.
- The close relationship between arrays and pointers makes a dynamically allocated array as easy to use as an ordinary array.
- Although `malloc` can allocate space for an array, the `calloc` function is sometimes used instead, since it initializes the memory that it allocates.
- The `realloc` function allows us to make an array "grow" or "shrink" as needed.

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 in C.
- 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;
```

The **calloc** Function

- The `calloc` function is an alternative to `malloc`.
- 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

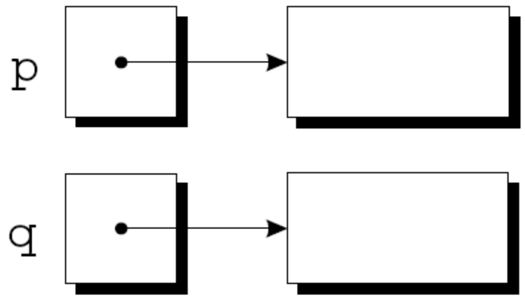
- `malloc` and the other memory allocation functions obtain memory blocks from a storage pool known as the ***heap***.
- Calling these functions too often—or asking them for large blocks of memory—can exhaust the heap, causing the functions to return a null pointer.
- To make matters worse, a program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

Deallocating Storage

- Example:

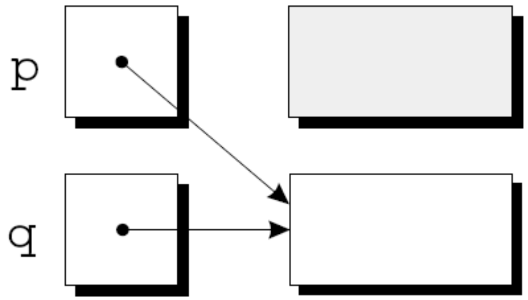
```
p = malloc(…);
q = malloc(…);
p = q;
```

- 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.