Introduction to Computers and Programming

Lecture 16 –

Structures, Unions, and Enumerations

Tien-Fu Chen

Dept. of Computer Science and Information Engineering

National Yang Ming Chiao Tung Univ.

Structures

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;
```

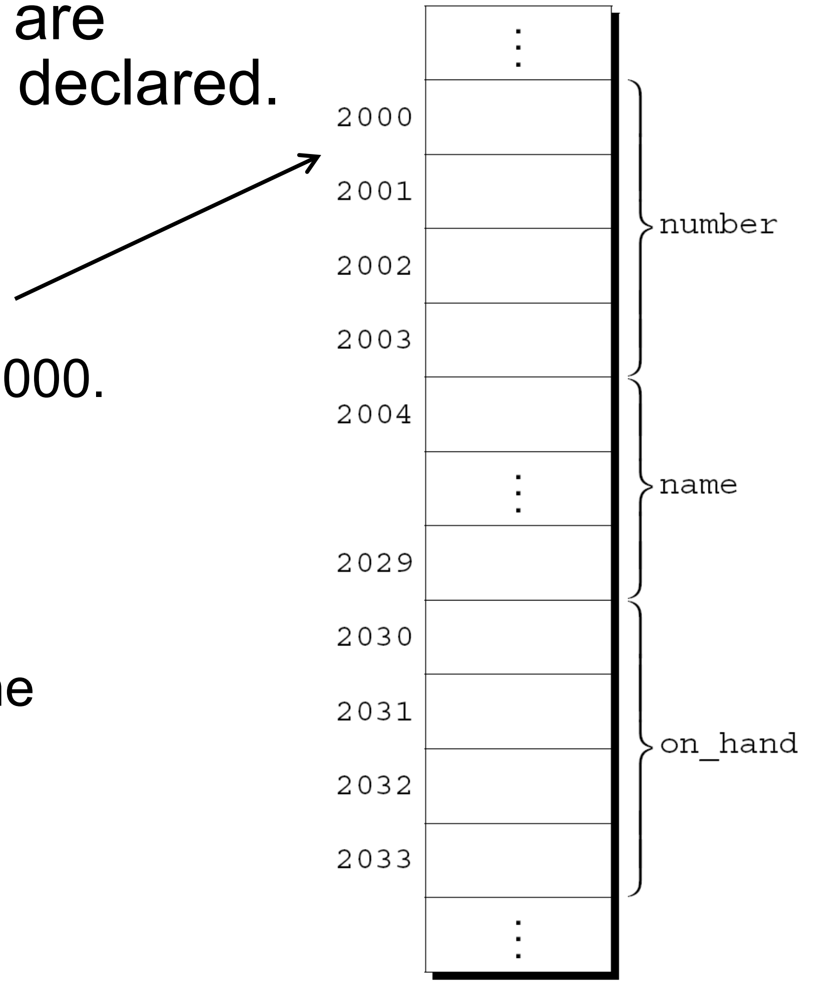
record

field

- The properties different from those of an array:
  - The elements of a structure (its members) aren't required to have the same type.

Declaring Structure Variables

- The members of a structure are stored in memory as 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

- Any names declared in that scope won't conflict with other names in a program:

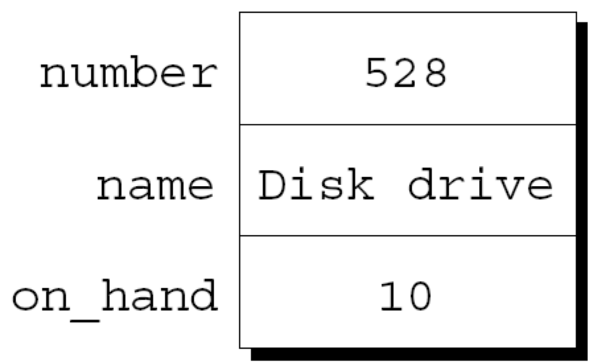
```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;

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:



Operations on Structures

- To access a member within a structure,
  - give the name of the structure first,
  - then a period,
  - then the name of the member.

- 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.
- Can appear on the left side of an assignment or
- Can do 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.

- Check sizeof(part1) ?

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.

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;
};
```

Structure tag

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

```
struct part 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 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 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.

- 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);
```

- Copying the information from a `person_name` structure to the `name` member of a `student` structure would take one assignment instead of three:

```
struct person_name new_name;
…
student1.name = new_name;
```

Arrays of Structures

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

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

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

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

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

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

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

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

Initializing an Array of Structures

- Initializing an array of structures is to store information that won’t change during program execution.

- 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}};
```

Unions

Unions: like a structure, overlay each other

- An example of a union variable:

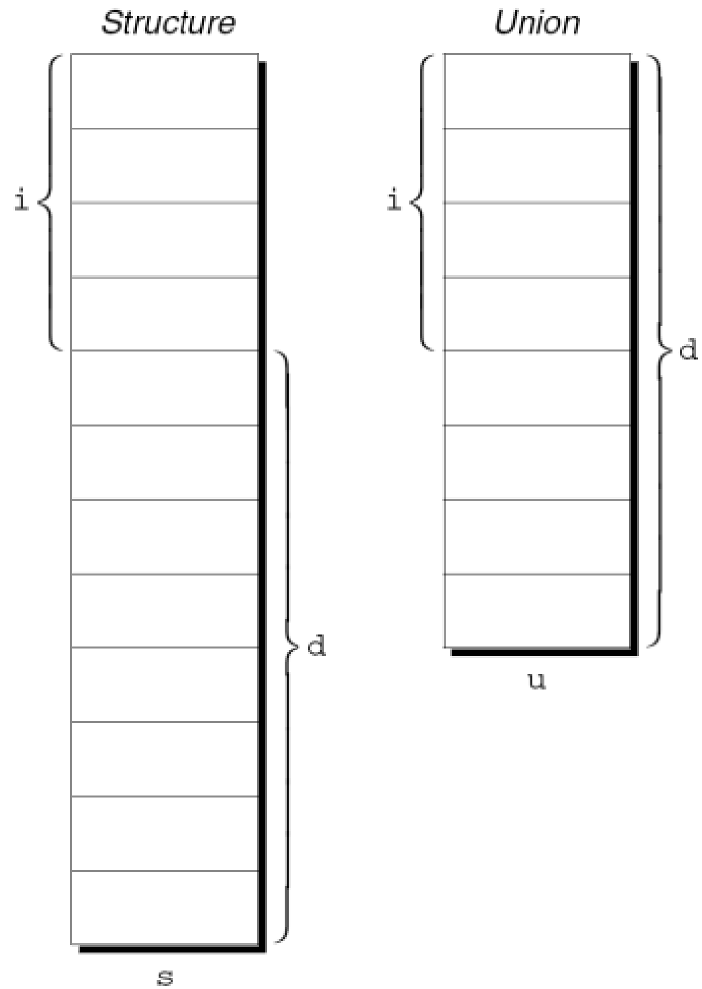
```
union {
  int i;
  double d;
} u;
```

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

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

Unions

- The members of `s` are stored separately in mmemory.
- 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

- 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.

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.

Using Unions to Save Space

- Unions can be used to save space in structures.
- Suppose that we're designing a structure that will contain information about an item that's sold through a gift catalog.
- Each item has a stock number and a price, as well as other information that depends on the type of the item:

  Books: Title, author, number of pages

  Mugs: Design

  Shirts: Design, colors available, sizes available

Using Unions to Save Space

- A first attempt at designing the `catalog_item` structure:

```
struct catalog_item {
  int stock_number;
  double price;
  int item_type;
  char title[TITLE_LEN+1];
  char author[AUTHOR_LEN+1];
  int num_pages;
  char design[DESIGN_LEN+1];
  int colors;
  int sizes;
};
```

Using Unions to Save Space

```
struct catalog_item {
  int stock_number;
  double price;
  int item_type;
  union {
    struct {
      char title[TITLE_LEN+1];
      char author[AUTHOR_LEN+1];
      int num_pages;
    } book;
    struct {
      char design[DESIGN_LEN+1];
    } mug;
    struct {
      char design[DESIGN_LEN+1];
      int colors;
      int sizes;
    } shirt;
  } item;
};
```

Using Unions to Save Space

- The union embedded in the `catalog_item` structure contains three structures as members.
- Two of these (`mug` and `shirt`) begin with a matching member (`design`).
- Now, suppose that we assign a value to one of the `design` members:

```
strcpy(c.item.mug.design, "Cats");
```

- The `design` member in the other structure will be defined and have the same value:

```
printf("%s", c.item.shirt.design);
  /* prints "Cats" */
```

Using Unions to Build Mixed Data Structures

- Unions can be used to create data structures that contain a mixture of data of different types.
- Suppose that we need an array whose elements are a mixture of `int` and `double` values.
- First, we define a union type whose members represent the two kinds of data to be stored in the array:

```
typedef union {
  int i;
  double d;
} Number;
```

Using Unions to Build Mixed Data Structures

- Next, we create an array whose elements are `Number` values:

```
Number number_array[1000];
```

- A `Number` union can store either an `int` value or a `double` value.
- This makes it possible to store a mixture of `int` and `double` values in `number_array`:

```
number_array[0].i = 5;
number_array[1].d = 8.395;
```

Adding a "Tag Field" to a Union

- The `Number` type as a structure with an embedded union:

```
#define INT_KIND 0
#define DOUBLE_KIND 1

typedef struct {
  int kind;   /* tag field */
  union {
    int i;
    double d;
  } u;
} Number;
```

- The value of `kind` will be either `INT_KIND` or `DOUBLE_KIND`.

Adding a "Tag Field" to a Union

- we assign a value to a member of `u`, we'll also change `kind` to indicate which member of `u` is modified.

- An example that assigns a value to the `i` member of `u`:

```
n.kind = INT_KIND;
n.u.i = 82;
```

  `n` is assumed to be a `Number` variable.

Adding a "Tag Field" to a Union

- When the number stored in a `Number` variable is retrieved, `kind` gives which member of the union was assigned.
- A function that takes advantage of this capability:

```
void print_number(Number n)
{
  if (n.kind == INT_KIND)
    printf("%d", n.u.i);
  else
    printf("%g", n.u.d);
}
```

Enumerations

Enumerations

- A variable that stores the suit of a playing card should have only four potential values: “clubs,” “diamonds,” “hearts,” and “spades.”
- A “suit” variable can be declared as an integer. A set of codes represent the possible values of the variable:

```
int s;   /* s will store a suit */
…
s = 2;   /* 2 represents "hearts" */
```

- Problems with this technique:
  - We can’t tell that `s` has only four possible values.
  - The significance of 2 isn’t apparent.

Enumerations

- Using macros to define a suit "type" and names for the various suits is a step in the right direction:

```
#define SUIT      int
#define CLUBS     0
#define DIAMONDS  1
#define HEARTS    2
#define SPADES    3
```

- An updated version of the previous example:

```
SUIT s;
…
s = HEARTS;
```

Enumerations

- An ***enumerated type*** is a type whose values are listed ("enumerated") by the programmer.
- Each value must have a name (an ***enumeration constant***).
- Although enumerations have little in common with structures and unions, they're declared in a similar way:

```
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s1, s2;
```

- The names of enumeration constants must be different from other identifiers declared in the enclosing scope.

Enumeration Tags and Type Names

- Two ways to name an enumeration: by declaring a tag or by using `typedef` to create a genuine type name.

- Enumeration tags resemble structure and union tags:

```
enum suit {CLUBS, DIAMONDS, HEARTS, SPADES};
```

- `suit` variables would be declared in the following way:

```
enum suit s1, s2;
```

Enumeration Tags and Type Names

- use `typedef` to make `Suit` a type name:

```
typedef enum {CLUBS, DIAMONDS, HEARTS, SPADES} Suit;
Suit s1, s2;
```

- In C89, using `typedef` to name an enumeration is an excellent way to create a Boolean type:

```
typedef enum {FALSE, TRUE} Bool;
```

Enumerations as Integers

- The programmer can choose different values for enumeration constants:

```
enum suit {CLUBS = 1, DIAMONDS = 2,
           HEARTS = 3, SPADES = 4};
```

- The values of enumeration constants may be arbitrary integers, listed in no particular order:

```
enum dept {RESEARCH = 20,
           PRODUCTION = 10, SALES = 25};
```

- It's even legal for two or more enumeration constants to have the same value.

Enumerations as Integers

- When no value is specified for an enumeration constant, its value is one greater than the value of the previous constant.
- The first enumeration constant has the value 0 by default.
- Example:

```
enum EGA_colors {BLACK, LT_GRAY = 7,
                DK_GRAY, WHITE = 15};
```

`BLACK` has the value 0, `LT_GRAY` is 7, `DK_GRAY` is 8, and `WHITE` is 15.

Enumerations as Integers

- Enumeration values can be mixed with ordinary integers:

```
int i;
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s;

i = DIAMONDS;    /* i is now 1              */
s = 0;           /* s is now 0 (CLUBS)      */
s++;             /* s is now 1 (DIAMONDS) */
i = s + 2;       /* i is now 3              */
```

- `s` is treated as a variable of some integer type.
- `CLUBS`, `DIAMONDS`, `HEARTS`, and `SPADES` are names for the integers 0, 1, 2, and 3.

Using Enumerations to Declare "Tag Fields"

- Enumerations are perfect for determining which member of a union was the last to be assigned.
- In the `Number` structure, we can make the `kind` member an enumeration instead of an `int`:

```
typedef struct {
  enum {INT_KIND, DOUBLE_KIND} kind;
  union {
    int i;
    double d;
  } u;
} Number;
```

Dynamic Storage Allocation

Dynamic Storage Allocation

- C's data structures, including arrays, are normally fixed in size.
- 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 is used most often for strings, arrays, and structures.
- Dynamic storage allocation is done by calling a memory allocation function. ***malloc() free()***

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.***
- 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 */
}
```

Null Pointers

- Pointers test true or false in the same way as numbers.
- All non-null pointers test true; only null pointers are false.
- Instead of writing

  ```
  if (p == NULL) …
  ```

  we could write

  ```
  if (!p) …
  ```

- Instead of writing

  ```
  if (p != NULL) …
  ```

  we could write

  ```
  if (p) …
  ```

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);
  ```

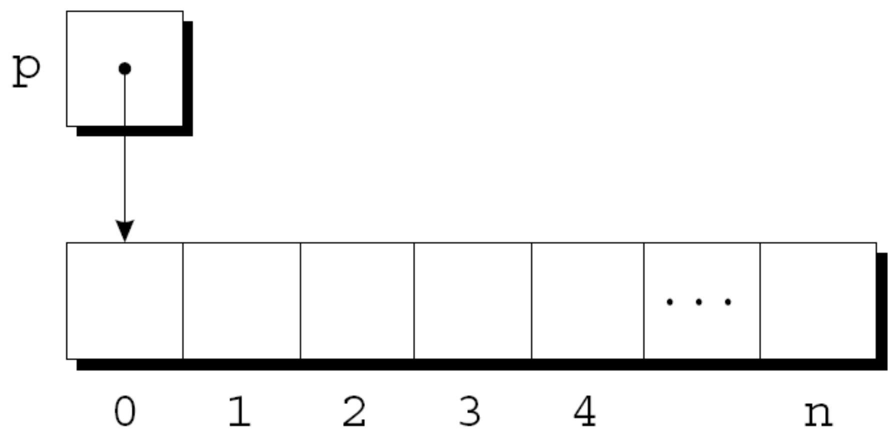
  `p` is a `char *` variable.

- Each character requires one byte of memory; adding 1 to `n` leaves room for the null character.
- Some programmers prefer to cast `malloc`'s return value, although the cast is not required:

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

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:

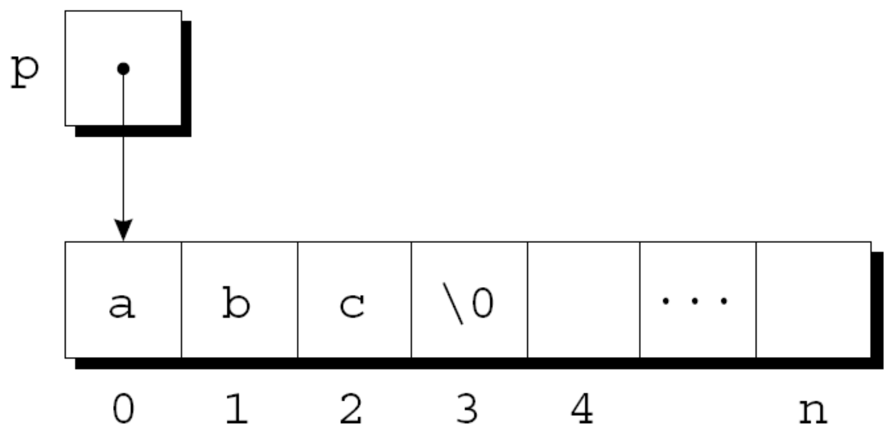


Using `malloc` to Allocate Memory for a String

- 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

```
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.

- When the string that `concat` returns is no longer needed, we'll want to call the `free` function to release.
- If we don't, the program may eventually run out of memory.

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.

```
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);
```

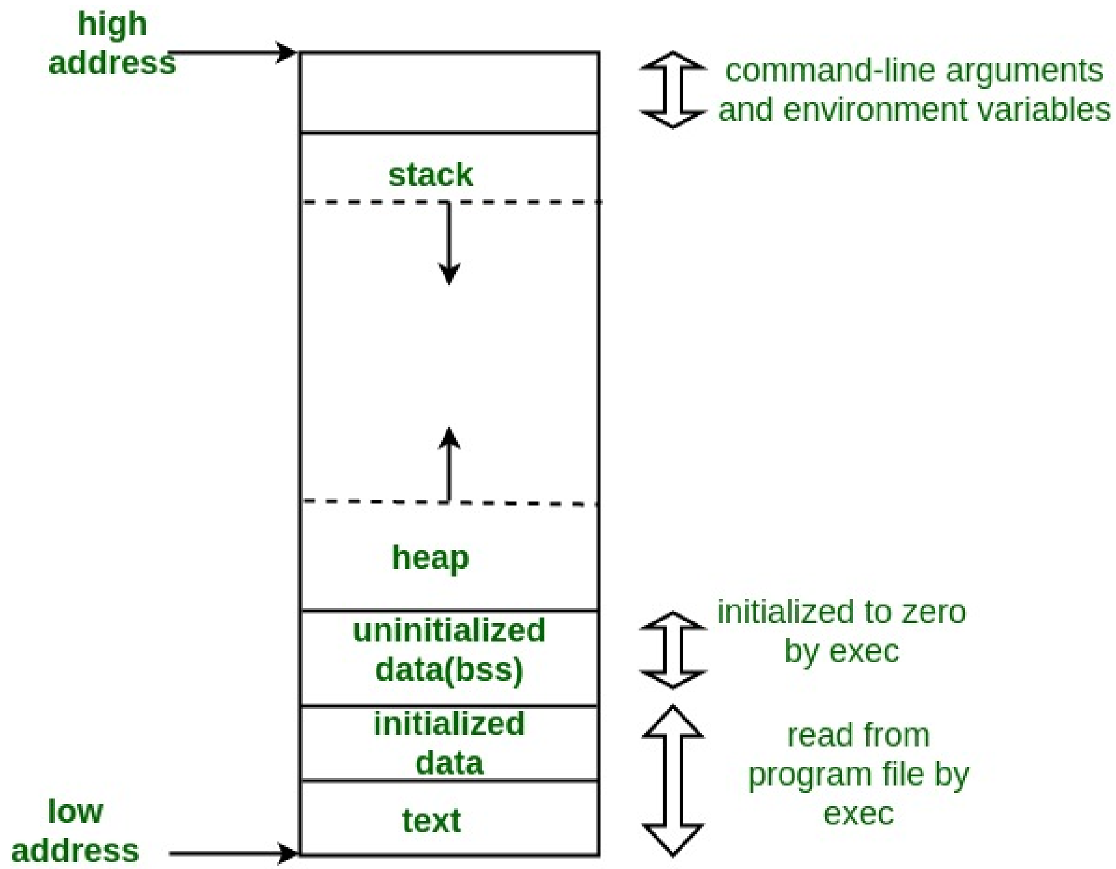
  - `ptr` must point to a memory block obtained by a previous call of `malloc`, `calloc`, or `realloc`.
  - `size` represents the new size of the block, which may be larger or smaller than the original size.

The `realloc` Function

- 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 ***heap.***
- Calling these functions too often—can exhaust the heap, causing the functions to return a null pointer.

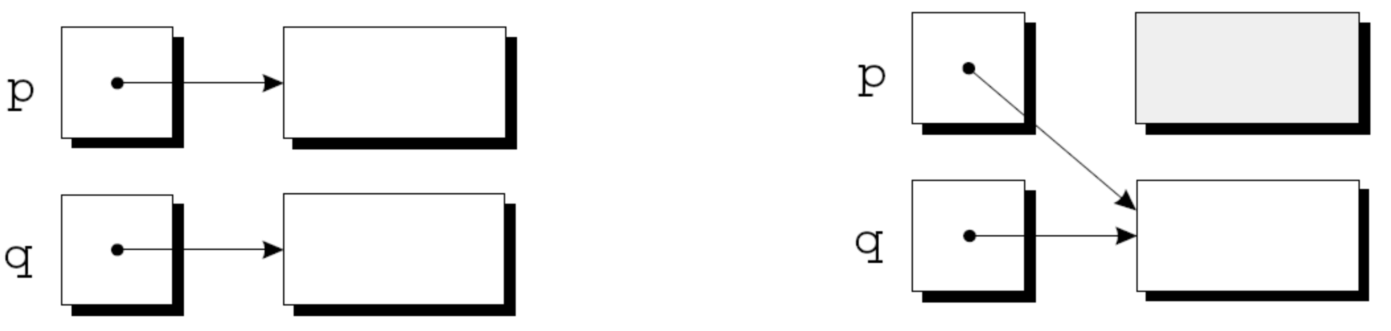


Deallocating Storage

- Example:

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

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



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.