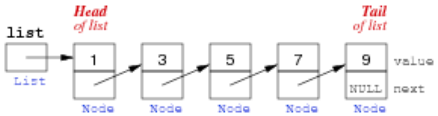
Week 3: Dynamic Data Structures

Pointers

Pointers

Reminder: In a *linked list* …

- each node contains a pointer to the next node
- the number of values can change dynamically



Benefits:

- insertion/deletion have minimal effect on list overall
- only use as much space as needed for values

In C, linked lists are implemented using *pointers* and *dynamic memory allocation*

Sidetrack: Numeral Systems

Numeral system … system for representing numbers using digits or other symbols.

- Most cultures have developed a *decimal* system (based on 10)
- For computers it is convenient to use a *binary* (base 2) or a *hexadecimal* (base 16) system

... Sidetrack: Numeral Systems

Decimal representation

- The base is 10; digits 0 - 9
- Example: decimal number 4705 can be interpreted as
$$4 \cdot 10^3 + 7 \cdot 10^2 + 0 \cdot 10^1 + 5 \cdot 10^0$$
- Place values:

…	1000	100	10	1
…	10^3	10^2	10^1	10^0

... Sidetrack: Numeral Systems

Binary representation

- The base is 2; digits 0 and 1
- Example: binary number 1101 can be interpreted as
$$1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$
- Place values:

…	8	4	2	1
…	2^3	2^2	2^1	2^0

- Write number as `0b1101` (= 13)

... Sidetrack: Numeral Systems

Hexadecimal representation

- The base is 16; digits 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- Example: hexadecimal number 3AF1 can be interpreted as
$$3 \cdot 16^3 + 10 \cdot 16^2 + 15 \cdot 16^1 + 1 \cdot 16^0$$
- Place values:

…	4096	256	16	1
…	16^3	16^2	16^1	16^0

- Write number as `0x3AF1` (= 15089)

Exercise #1: Conversion Between Different Numeral Systems

1. Convert `74` to base 2
2. Convert `0x2D` to base 10
3. Convert `0b1011111000101001` to base 16
   - Hint: `1011111000101001`
4. Convert `0x12D` to base 2

1. `0b1001010`
2. `45`
3. `0xBE29`
4. `0b100101101`

Memory

Computer memory … large array of consecutive data cells or bytes

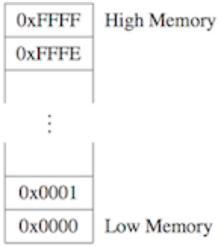
- `char` … 1 byte `int,float` … 4 bytes `double` … 8 bytes

When a variable is declared, the operating system finds a place in memory to store the appropriate number of bytes.

If we declare a variable called `k` ...

- the place where `k` is stored is denoted by `&k`
- also called the address of k

It is convenient to print memory addresses in Hexadecimal notation



... Memory

Example:

```
int k;
int m;

printf("address of k is %p\n", &k);
printf("address of m is %p\n", &m);
```

```
address of k is BFFFFB80
address of m is BFFFFB84
```

This means that

- `k` occupies the four bytes from `BFFFFB80` to `BFFFFB83`
- `m` occupies the four bytes from `BFFFFB84` to `BFFFFB87`

Note the use of `%p` as placeholder for an address ("pointer" value)

... Memory

When an array is declared, the elements of the array are guaranteed to be stored in consecutive memory locations:

```
int array[5];

for (i = 0; i < 5; i++) {
   printf("address of array[%d] is %p\n", i, &array[i]);
}
```

```
address of array[0] is BFFFFB60
address of array[1] is BFFFFB64
address of array[2] is BFFFFB68
address of array[3] is BFFFFB6C
address of array[4] is BFFFFB70
```

Application: Input Using `scanf()`

Standard I/O function `scanf()` requires the *address* of a variable as argument

- `scanf()` uses a format string like `printf()`
- use `%d` to read an integer value

```
#include <stdio.h>
…
int answer;
printf("Enter your answer: ");
scanf("%d", &answer);
```

- use `%f` to read a floating point value (`%lf` for `double`)

```
float e;
printf("Enter e: ");
scanf("%f", &e);
```

- `scanf()` returns a value — the number of items read
  - use this value to determine if `scanf()` successfully read a number
    - `scanf()` could fail e.g. if the user enters letters

Exercise #2: Using `scanf`

Write a program that

- asks the user for a number
- checks that it is positive
- applies Collatz's process (Exercise 3, Problem Set Week 1) to the number

---

```
#include <stdio.h>

void collatz(int n) {
   printf("%d\n", n);
   while (n != 1) {
      if (n % 2 == 0)
         n = n / 2;
      else
         n = 3*n + 1;
      printf("%d\n", n);
   }
}

int main(void) {
   int n;
   printf("Enter a positive number: ");
   if (scanf("%d", &n) == 1 && (n > 0))   /* test if scanf successful
                                             and returns positive number */
      collatz(n);
   return 0;
}
```

Pointers

A *pointer* …

- is a special type of variable
- storing the address (memory location) of another variable

A pointer occupies space in memory, just like any other variable of a certain type

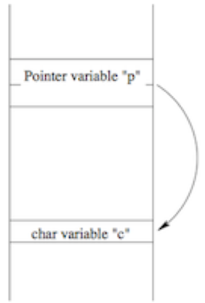
The number of memory cells needed for a pointer depends on the computer's architecture:

- Old computer, or hand-held device with only 64KB of addressable memory:
  - 2 memory cells (i.e. 16 bits) to hold any address from `0x0000` to `0xFFFF` (= 65535)
- Desktop machine with 4GB of addressable memory
  - 4 memory cells (i.e. 32 bits) to hold any address from `0x00000000` to `0xFFFFFFFF` (= 4294967295)
- Modern 64-bit computer
  - 8 memory cells (can address 2^{64} bytes, but in practice the amount of memory is limited by the CPU)

... Pointers

Suppose we have a pointer **p** that "points to" a `char` variable `c`.

Assuming that the pointer **p** requires 2 bytes to store the address of `c`, here is what the memory map might look like:



... Pointers

Now that we have assigned to p the address of variable `c` …

- need to be able to reference the data in that memory location

Operator `*` is used to access the object the pointer points to

- e.g. to change the value of `c` using the pointer `p`:

```
*p = 'T';  // sets the value of c to 'T'
```

The `*` operator is sometimes described as "*dereferencing*" the pointer, to access the underlying variable

---

... Pointers

Things to note:

- all pointers constrained to point to a particular type of object

```
// a potential pointer to any object of type char
char *s;

// a potential pointer to any object of type int
int *p;
```

- if pointer `p` is pointing to an integer variable `x`
  ⇒ `*p` can occur in any context that `x` could

Examples of Pointers


```
int *p; int *q; // this is how pointers are declared
int a[5];
int x = 10, y;

 p = &x;      // p now points to x
*p = 20;      // whatever p points to is now equal to 20
 y = *p;      // y is now equal to whatever p points to
 p = &a[2];   // p points to an element of array a[]
 q = p;       // q and p now point to the same thing
```

Exercise #3: Pointers

What is the output of the following program?

```
#include <stdio.h>

int main(void) {
   int *ptr1, *ptr2;
   int i = 10, j = 20;

   ptr1 = &i;
   ptr2 = &j;

   *ptr1 = *ptr1 + *ptr2;
   ptr2 = ptr1;
   *ptr2 = 2 * (*ptr2);
   printf("Val = %d\n", *ptr1 + *ptr2);
   return 0;
}
```

`Val = 120`

... Examples of Pointers

22/96

Can we write a function to "swap" two variables?

The *wrong* way:

```
void swap(int a, int b) {
   int temp = a;                     // only local "copies" of a and b will swap
   a = b;
   b = temp;
}

int main(void) {
   int a = 5, b = 7;
   swap(a, b);
   printf("a = %d, b = %d\n", a, b); // a and b still have their original values
   return 0;
}
```

... Examples of Pointers

23/96

In C, parameters are "call-by-value"

- changes made to the value of a parameter do not affect the original
- function `swap()` tries to swap the values of `a` and `b`, but fails because it only swaps the copies, not the "real" variables in `main()`

We can achieve "simulated call-by-reference" by passing pointers as parameters

- this allows the function to change the "actual" value of the variables

... Examples of Pointers

24/96

Can we write a function to "swap" two variables?

The *right* way:

```
void swap(int *p, int *q) {
   int temp = *p;                  // change the actual values of a and b
   *p = *q;
   *q = temp;
}

int main(void) {
   int a = 5, b = 7;
   swap(&a, &b);
   printf("a = %d, b = %d\n", a, b);  // a and b now successfully swapped
   return 0;
}
```

Pointer Arithmetic

25/96

A *pointer* variable holds a value which is an *address*.

C knows what type of object is being pointed to

- it knows the `sizeof` that object
- it can compute where the next/previous object is located

Example:

```
int a[6];   // assume array starts at address 0x1000
int *p;
p = &a[0];  // p contains 0x1000
p = p + 1;  // p now contains 0x1004
```

... Pointer Arithmetic

26/96

For a pointer declared as `T *p;` (where `T` is a type)

- if the pointer initially contains address `A`
  - executing `p = p + k;` (where k is a constant)
    - changes the value in `p` to `A + k*sizeof(T)`

The value of `k` can be positive or negative.

Example:

```
int a[6];   (addr 0x1000)       char s[10];   (addr 0x2000)
int *p;     (p == ?)           char *q;      (q == ?)
p = &a[0];  (p == 0x1000)       q = &s[0];    (q == 0x2000)
p = p + 2;  (p == 0x1008)       q++;          (q == 0x2001)
```

Pointers and Arrays

27/96

An alternative approach to iteration through an array:

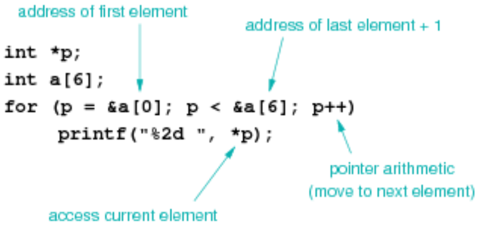
- determine the address of the first element in the array
- determine the address of the last element in the array
- set a pointer variable to refer to the first element
- use pointer arithmetic to move from element to element
- terminate loop when address exceeds that of last element

Example:

```
int a[6];
int *p;
p = &a[0];
while (p <= &a[5]) {
    printf("%2d ", *p);
    p++;
}
```

... Pointers and Arrays

Pointer-based scan written in more typical style



Note: because of pointer/array connection `a[i] == *(a+i)`

Arrays of Strings

One common type of pointer/array combination are the *command line arguments*

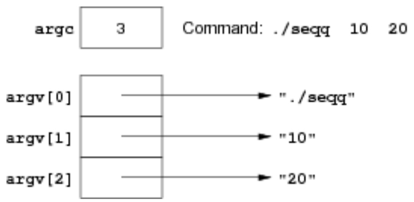
- These are 0 or more strings specified when program is run
- Suppose you have an excutable program named `seqq`. If you run this command in a terminal:

  ```
  prompt$ ./seqq 10 20
  ```

  then `seqq` will be given 2 command-line arguments: `"10", "20"`

... Arrays of Strings


```
prompt$ ./seqq 10 20
```



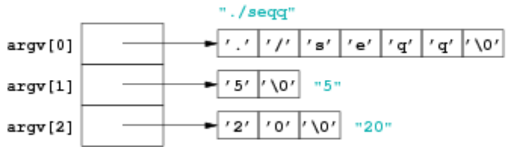
Each element of `argv[]` is

- a pointer to the start of a character array (`char *`)
  - containing a `\0`-terminated string

... Arrays of Strings

More detail on how `argv` is represented:

```
prompt$ ./seqq 5 20
```



... Arrays of Strings

`main()` needs different prototype if you want to access command-line arguments:

```
int main(int argc, char *argv[]) { ...
```

- **`argc`** ... stores the number of command-line arguments + 1
  - `argc == 1` if no command-line arguments
- **`argv[]`** ... stores program name + command-line arguments
  - `argv[0]` always contains the program name
  - `argv[1],argv[2],...` are the command-line arguments if supplied

`<stdlib.h>` defines useful functions to convert strings:

- **`atoi(char *s)`** converts string to int
- **`atof(char *s)`** converts string to double (can also be assigned to `float` variable)

Exercise #4: Command Line Arguments

Write a program that

- checks for a single command line argument
  - if not, outputs a usage message and exits with failure
- converts this argument to a number and checks that it is positive
- applies Collatz's process (Exercise 3, Problem Set Week 1) to the number

```
#include <stdio.h>
#include <stdlib.h>

void collatz(int n) {
   ...
}

int main(int argc, char *argv[]) {
   if (argc != 2) {
      printf("Usage: %s number\n", argv[0]);
      return 1;
```

```
   }
   int n = atoi(argv[1]);
   if (n > 0)
      collatz(n);
   return 0;
}
```

... Arrays of Strings

35/96

argv can also be viewed as *double pointer* (a pointer to a pointer)

⇒ Alternative prototype for `main()`:

```
int main(int argc, char **argv) { ...
```

Can still use `argv[0]`, `argv[1]`, …

Pointers and Structures

36/96

Like any object, we can get the address of a `struct` via `&`.

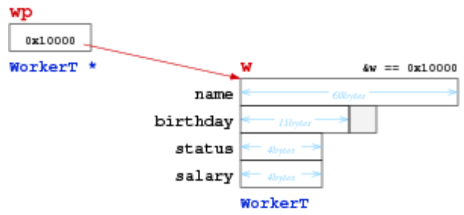
```
typedef char Date[11];  // e.g. "03-08-2017"
typedef struct {
    char  name[60];
    Date  birthday;
    int   status;      // e.g. 1 (≡ full time)
    float salary;
} WorkerT;

WorkerT w;  WorkerT *wp;
wp = &w;
// a problem ...
*wp.salary = 125000.00;
// does not have the same effect as
w.salary = 125000.00;
// because it is interpreted as
*(wp.salary) = 125000.00;

// to achieve the correct effect, we need
(*wp).salary = 125000.00;
// a simpler alternative is normally used in C
wp->salary = 125000.00;
```

Learn this well; we will frequently use it in this course.

... Pointers and Structures

37/96

Diagram of scenario from program above:



... Pointers and Structures

38/96

General principle …

If we have:

```
SomeStructType s;
SomeStructType *sp = &s;  // declare pointer and initialise to address of s
```

then the following are all equivalent:

`s.SomeElem`   `sp->SomeElem`   `(*sp).SomeElem`

Memory

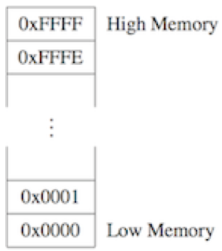
39/96

Reminder:

Computer memory … large array of consecutive data cells or bytes

- `char` … 1 byte
- `int`,`float` … 4 bytes
- `double` … 8 bytes
- *`any_type *`* … 8 bytes (on CSE lab computers)

Memory addresses shown in Hexadecimal notation



C execution: Memory

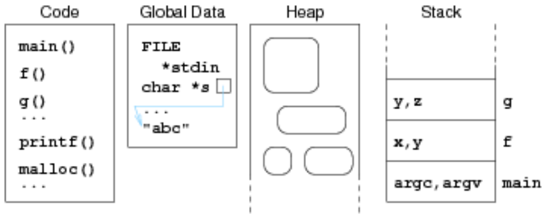
40/96

An executing C program partitions memory into:

- *code* … fixed-size, read-only region
    - contains the machine code instructions for the program

- *global data* … fixed-size
  - contain global variables (read-write) and constant strings (read-only)
- *heap* … very large, read-write region
  - contains dynamic data structures created by `malloc()` (see later)
- *stack* … dynamically-allocated data (function local vars)
  - consists of frames, one for each currently active function
  - each frame contains local variables and house-keeping info

... C execution: Memory



Exercise #5: Memory Regions


```
int numbers[] = { 40, 20, 30 };

void insertionSort(int array[], int n) {
   int i, j;
   for (i = 1; i < n; i++) {
      int element = array[i];
      for (j = i-1; j >= 0 && array[j] > element; j--)
         array[j+1] = array[j];
      array[j+1] = element;
   }
}

int main(void) {
   insertionSort(numbers, 3);
   return 0;
}
```

Which memory region are the following objects located in?

1. `insertionSort()`
2. `numbers[0]`
3. `n`
4. `array[0]`
5. `element`

1. code
2. global
3. stack
4. global
5. stack

Dynamic Data Structures

Dynamic Memory Allocation

So far, we have considered *static* memory allocation

- all objects completely defined at compile-time
- sizes of all objects are known to compiler

Examples:

```
int   x;     // 4 bytes containing a 32-bit integer value
char *cp;    // 8 bytes (on CSE machines)
             //   containing address of a char
typedef struct {float x; float y;} Point;
Point p;     // 8 bytes containing two 32-bit float values
char  s[20]; // array containing space for 20 1-byte chars
```

... Dynamic Memory Allocation

In many applications, fixed-size data is ok.

In many other applications, we need flexibility.

Examples:

```
char name[MAXNAME];   // how long is a name?
char item[MAXITEMS];  // how high can the stack grow?
char dictionary[MAXWORDS][MAXWORDLENGTH];
                     // how many words are there?
                     // how long is each word?
```

With fixed-size data, we need to guess sizes ("large enough").

... Dynamic Memory Allocation

Fixed-size memory allocation:

- allocate as much space as we might ever possibly need

Dynamic memory allocation:

- allocate as much space as we actually need
- determine size based on inputs

But how to do this in C?

- all data allocation methods so far are "static"
  - however, stack data (when calling a function) is created dynamically (size is known)

Dynamic Data Example

Problem:

- read integer data from standard input (keyboard)
- first number tells how many numbers follow
- rest of numbers are read into a vector
- subsequent computation uses vector (e.g. sorts it)

Example input: `6 25 -1 999 42 -16 64`

How to define the vector?

... Dynamic Data Example

Suggestion #1: allocate a large vector; use only part of it

```
#define MAXELEMS 1000

// how many elements in the vector
int numberOfElems;
scanf("%d", &numberOfElems);
assert(numberOfElems <= MAXELEMS);

// declare vector and fill with user input
int i, vector[MAXELEMS];
for (i = 0; i < numberOfElems; i++)
   scanf("%d", &vector[i]);
```

Works ok, unless too many numbers; usually wastes space.

Recall that `assert()` terminates program with standard error message if test fails.

... Dynamic Data Example

Suggestion #2: create vector after count read in

```
#include <stdlib.h>

// how many elements in the vector
int numberOfElems;
scanf("%d", &numberOfElems);

// declare vector and fill with user input
int i, *vector;
size_t numberOfBytes;
numberOfBytes = numberOfElems * sizeof(int);

vector = malloc(numberOfBytes);
assert(vector != NULL);
```

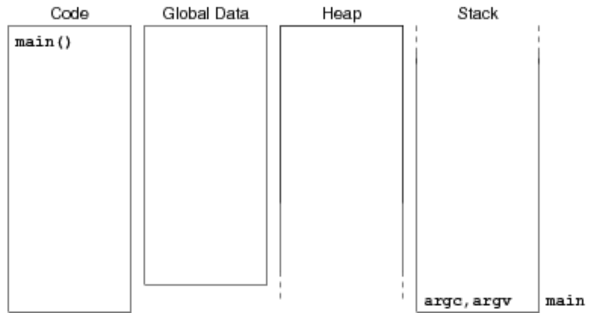
```
for (i = 0; i < numberOfElems; i++)
   scanf("%d", &vector[i]);
```

Works unless the *heap* is already full (very unlikely)

Reminder: because of pointer/array connection `&vector[i] == vector+i`

The `malloc()` function

Recall memory usage within C programs:



... The `malloc()` function

`malloc()` function interface

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

What the function does:

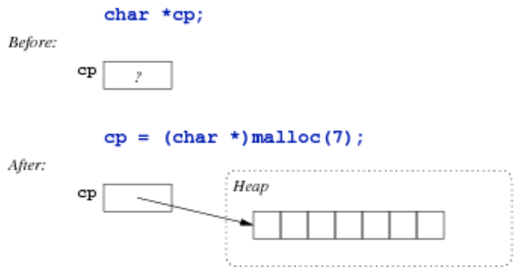
- attempts to reserve a block of `n` bytes in the *heap*
- returns the address of the start of this block
- if insufficient space left in the heap, returns `NULL`

Note: `size_t` is essentially an `unsigned int`

- but has specialised interpretation of applying to memory sizes measured in bytes

... The `malloc()` function

Example use of `malloc`:



Note: because of `a[i] == *(a+i)` can use `cp[i]` to refer to i-th element of the dynamic array

---

... The `malloc()` function

54/96

Things to note about `void *malloc(size_t)`:

- it is defined as part of `stdlib.h`
- its parameter is a size in units of *bytes*
- its return value is a *generic* pointer `(void *)`
- the return value must *always* be checked (may be `NULL`)

Required size is determined by *#Elements* `*` `sizeof(`*ElementType*`)`

Exercise #6: Dynamic Memory Allocation

55/96

Write code to

1. create space for 1,000 speeding tickets (cf. Lecture Week 1)
2. create a dynamic $m \times n$-matrix of floating point numbers, given m and n

How many bytes need to be reserved in each case?

---

1. Speeding tickets:

```
typedef struct {
        int day, month; } DateT;
typedef struct {
        int hour, minute; } TimeT;
typedef struct {
        char plate[7]; double speed; DateT d; TimeT t; } TicketT;

TicketT *tickets;
tickets = malloc(1000 * sizeof(TicketT));
assert(tickets != NULL);
```

32,000 bytes allocated

2. Matrix:

```
float **matrix;

// allocate memory for m pointers to beginning of rows
matrix = malloc(m * sizeof(float *));
assert(matrix != NULL);

// allocate memory for the elements in each row
int i;
for (i = 0; i < m; i++) {
   matrix[i] = malloc(n * sizeof(float));
   assert(matrix[i] != NULL);
}
```

$8m + 4 \cdot mn$ bytes allocated

Exercise #7: Memory Regions

57/96

Which memory region is `tickets` located in? What about `*tickets`?

---

1. `tickets` is a variable located in the stack
2. `*tickets` is in the heap (after **`malloc`**'ing memory)

... The `malloc()` function

59/96

`malloc()` returns a pointer to a data object of some kind.

Things to note about objects allocated by `malloc()`:

- they exist until explicitly removed (program-controlled lifetime)
- they are *accessible* while some variable references them
- if no active variable references an object, it is *garbage*

The function `free()` releases objects allocated by `malloc()`

... The `malloc()` function

60/96

Usage of `malloc()` should always be guarded:

```
int *vector, length, i;
…
vector = malloc(length*sizeof(int));
// but malloc() might fail to allocate
assert(vector != NULL);
// now we know it's safe to use vector[]
for (i = 0; i < length; i++) {
        … vector[i] …
}
```

Alternatively:

```
int *vector, length, i;
…
vector = malloc(length*sizeof(int));
// but malloc() might fail to allocate
if (vector == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(1);
}
// now we know its safe to use vector[]
for (i = 0; i < length; i++) {
        … vector[i] …
}
```

- `fprintf(stderr, …)` outputs text to a stream called **`stderr`** (the screen, by default)
- `exit(v)` terminates the program with return value *v*

Memory Management

`void free(void *ptr)`

- releases a block of memory allocated by `malloc()`
- `*ptr` is a dynamically allocated object
- if `*ptr` was not `malloc()`'d, chaos will follow

Things to note:

- the contents of the memory block are not changed
- all pointers to the block still exist, but are not valid
- the memory may be re-used as soon as it is `free()`'d

... Memory Management

Warning! Warning! Warning! Warning!

Careless use of `malloc()` / `free()` / pointers

- can mess up the data in the heap
- so that later `malloc()` or `free()` cause run-time errors
- possibly well after the original error occurred

Such errors are very difficult to track down and debug.

Must be very careful with your use of `malloc()` / `free()` / pointers.

... Memory Management

If an uninitialised or otherwise invalid pointer is used, or an array is accessed with a negative or out-of-bounds index, one of a number of things might happen:

- program aborts immediately with a "`segmentation fault`"
- a mysterious failure much later in the execution of the program
- incorrect results, but no obvious failure
- correct results, but maybe not always, and maybe not when executed on another day, or another machine

The first is the most desirable, but cannot be relied on.

... Memory Management

Given a pointer variable:

- you can check whether its value is `NULL`
- you can (maybe) check that it is an address
- you **cannot** check whether it is a valid address

... Memory Management

Typical usage pattern for dynamically allocated objects:

```
// single dynamic object e.g. struct
Type *ptr = malloc(sizeof(Type));  // declare and initialise
assert(ptr != NULL);
… use object referenced by ptr e.g. ptr->name …
free(ptr);

// dynamic array with "nelems" elements
int nelems = NumberOfElements;
ElemType *arr = malloc(nelems*sizeof(ElemType));
assert(arr != NULL);
… use array referenced by arr e.g. arr[4] …
free(arr);
```

Memory Leaks

Well-behaved programs do the following:

- allocate a new object via `malloc()`
- use the object for as long as needed
- `free()` the object when no longer needed

A program which does not `free()` each object before the last reference to it is lost contains a *memory leak*.

Such programs may eventually exhaust available heapspace.

Exercise #8: Dynamic Arrays

Write a C-program that

- prompts the user to input a positive number *n*
- allocates memory for two *n*-dimensional floating point vectors **a** and **b**
- prompts the user to input 2*n* numbers to initialise these vectors
- computes and outputs the inner product of **a** and **b**
- frees the allocated memory

Sidetrack: Standard I/O Streams, Redirects

Standard file streams:

- **`stdin`** … standard input, by default: keyboard
- **`stdout`** … standard output, by default: screen
- **`stderr`** … standard error, by default: screen

- `fprintf(stdout, …)` has the same effect as `printf(…)`
- `fprintf(stderr, …)` often used to print error messages

Executing a C program causes `main(…)` to be invoked

- with `stdin`, `stdout`, `stderr` already open for use

... Sidetrack: Standard I/O Streams, Redirects

The streams `stdin`, `stdout`, `stderr` can be *redirected*

- redirecting `stdin`

```
prompt$ myprog < input.data
```

- redirecting `stdout`

```
prompt$ myprog > output.data
```

- redirecting `stderr`

```
prompt$ myprog 2> error.data
```

Linked Lists as Dynamic Data Structure

Sidetrack: Defining Structures

Structures can be defined in two different styles:

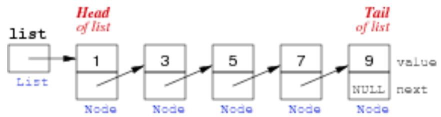
```
typedef struct { int day, month, year; } DateT;
// which would be used as
DateT somedate;

// or
struct date { int day, month, year; };
// which would be used as
struct date anotherdate;
```

The definitions produce objects with identical structures.

It is possible to combine both styles:

```
typedef struct date { int day, month, year; } DateT;
// which could be used as
DateT       date1, *dateptr1;
struct date date2, *dateptr2;
```

Self-referential Structures



Reminder: To realise a "chain of elements", need a *node* containing

- a value
- a link to the next node

In C, we can define such nodes as:

```
typedef struct node {
   int data;
   struct node *next;
} NodeT;
```

... Self-referential Structures

Note that the following definition does not work:

```
typedef struct {
   int data;
   NodeT *next;
} NodeT;
```

Because `NodeT` is not yet known (to the compiler) when we try to use it to define the type of the `next` field.

The following is also illegal in C:

```
struct node {
   int data;
   struct node recursive;
};
```

Because the size of the structure would have to satisfy `sizeof(struct node) = sizeof(int) + sizeof(struct node) = ∞`.

Memory Storage for Linked Lists

74/96

Linked list nodes are typically located in the heap

- because nodes are dynamically created

Variables containing pointers to list nodes

- are likely to be local variables (in the stack)

Pointers to the start of lists are often

- passed as parameters to function
- returned as function results

... Memory Storage for Linked Lists

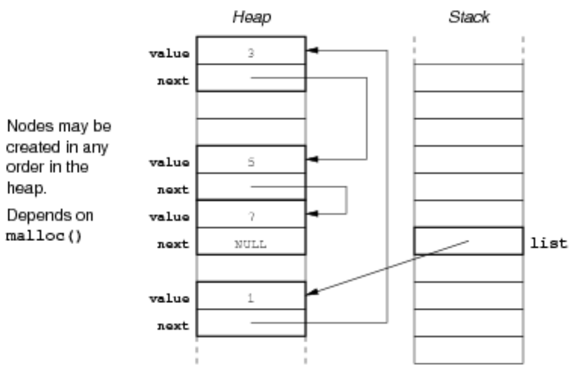
75/96

Create a new list node:

```
NodeT *makeNode(int v) {
   NodeT *new = malloc(sizeof(NodeT));
   assert(new != NULL);
   new->data = v;       // initialise data
   new->next = NULL;    // initialise link to next node
   return new;          // return pointer to new node
}
```

... Memory Storage for Linked Lists

76/96



Iteration over Linked Lists

77/96

When manipulating list elements

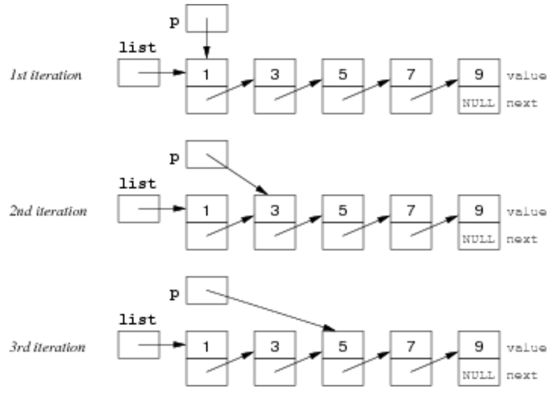
- typically have pointer p to current node (NodeT *p)
- to access the data in current node: p->data
- to get pointer to next node: p->next

To iterate over a linked list:

- set p to point at first node (head)
- examine node pointed to by p
- change p to point to next node
- stop when p reaches end of list (NULL)

... Iteration over Linked Lists

78/96



... Iteration over Linked Lists

79/96

Standard method for scanning all elements in a linked list:

```
NodeT *list;  // pointer to first Node in list
NodeT *p;     // pointer to "current" Node in list

p = list;
while (p != NULL) {
        … do something with p->data …
        p = p->next;
}

// which is frequently written as

for (p = list; p != NULL; p = p->next) {
        … do something with p->data …
}
```

... Iteration over Linked Lists

80/96

Check if list contains an element:

```
int inLL(NodeT *list, int d) {
   NodeT *p;
   for (p = list; p != NULL; p = p->next)
      if (p->data == d)      // element found
         return true;
   return false;             // element not in list
}
```

Print all elements:

```
void showLL(NodeT *list) {
   NodeT *p;
   for (p = list; p != NULL; p = p->next)
      printf("%6d", p->data);
}
```

Modifying a Linked List

81/96

Insert a new element at the beginning:

```
NodeT *insertLL(NodeT *list, int d) {
   NodeT *new = makeNode(d);  // create new list element
   new->next = list;          // link to beginning of list
   return new;                // new element is new head
}
```

Delete the first element:

```
NodeT *deleteHead(NodeT *list) {
   assert(list != NULL);  // ensure list is not empty
   NodeT *head = list;    // remember address of first element
   list = list->next;     // move to second element
   free(head);
   return list;           // return pointer to second element
}
```

What would happen if we didn't `free` the memory pointed to by `head`?

Exercise #9: Freeing a list

82/96

Write a C-function to destroy an entire list.

Iterative version:

```
void freeLL(NodeT *list) {
   NodeT *p, *temp;

   p = list;
   while (p != NULL) {
      temp = p->next;
      free(p);
      p = temp;
   }
}
```

Why do we need the extra variable `temp`?

Abstract Data Structures: ADTs

Abstract Data Types

85/96

Reminder: An *abstract data type* is …

- an approach to implementing data types
- separates *interface* from *implementation*
- users of the ADT see only the interface
- builders of the ADT provide an implementation

E.g. does a client want/need to know how a Stack is implemented?

- ADO = *a*bstract *d*ata *o*bject (e.g. a single stack)
- ADT = *a*bstract *d*ata *type* (e.g. stack data type)

... Abstract Data Types

86/96

Typical operations with ADTs

- *create* a value of the type
- *modify* one variable of the type
- *combine* two values of the type

... Abstract Data Types

87/96

ADT *interface* provides

- an *opaque* user-view of the data structure (e.g. `stack *`)
- function signatures (prototypes) for all operations
- semantics of operations (via documentation)
- a contract between ADT and its clients

ADT *implementation* gives

- concrete definition of the data structure
- function implementations for all operations
- … including for *creation* and *destruction* of instances of the data structure

ADTs are important because …

- facilitate decomposition of complex programs
- make implementation changes invisible to clients

- improve readability and structuring of software

Stack as ADT

88/96

Interface (in `stack.h`)

```
// provides an opaque view of ADT
typedef struct StackRep *stack;

// set up empty stack
stack newStack();
// remove unwanted stack
void dropStack(stack);
// check whether stack is empty
int StackIsEmpty(stack);
// insert an int on top of stack
void StackPush(stack, int);
// remove int from top of stack
int StackPop(stack);
```

ADT *`stack`* defined as a *pointer* to an *unspecified* struct named `StackRep`

Stack ADT Implementation

89/96

Linked list implementation (`stack.c`):

Remember: `stack.h` includes `typedef struct StackRep *stack;`

```
#include <stdlib.h>
#include <assert.h>
#include "stack.h"

typedef struct node {
   int data;
   struct node *next;
} NodeT;

typedef struct StackRep {
   int    height;   // #elements on stack
   NodeT *top;      // ptr to first element
} StackRep;

// set up empty stack
stack newStack() {
   stack S = malloc(sizeof(StackRep));
   S->height = 0;
   S->top = NULL;
   return S;
}

// remove unwanted stack
void dropStack(stack S) {
   NodeT *curr = S->top;
   while (curr != NULL) {  // free the list
      NodeT *temp = curr->next;
      free(curr);
      curr = temp;
   }
```

```
// check whether stack is empty
int StackIsEmpty(stack S) {
   return (S->height == 0);
}

// insert an int on top of stack
void StackPush(stack S, int v) {
   NodeT *new = malloc(sizeof(NodeT));
   assert(new != NULL);
   new->data = v;
   // insert new element at top
   new->next = S->top;
   S->top = new;
   S->height++;
}

// remove int from top of stack
int StackPop(stack S) {
   assert(S->height > 0);
   NodeT *head = S->top;
   // second list element becomes new top
   S->top = S->top->next;
   S->height--;
   // read data off first element, then free
   int d = head->data;
```

```
   free(S);           // free the stack rep
}
```

```
   free(head);
   return d;
}
```

Sidetrack: Make/Makefiles

90/96

Compilation process is complex for large systems.

How much to compile?

- ideally, what's changed since last compile
- practically, recompile everything, to be sure

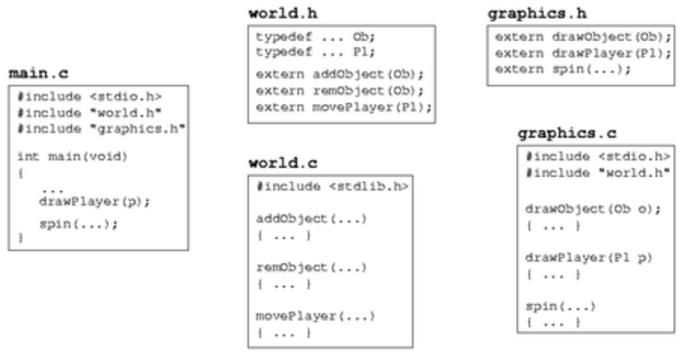
The **`make`** command assists by allowing

- programmers to document *dependencies* in code
- minimal re-compilation, based on dependencies

... Sidetrack: Make/Makefiles

91/96

Example multi-module program …



... Sidetrack: Make/Makefiles

92/96

`make` is driven by dependencies given in a **`Makefile`**

A *dependency* specifies

`target` `:` *`source`*$_1$ *`source`*$_2$ …
        `commands to build target from sources`

e.g.

```
game : main.o graphics.o world.o
```

```
        gcc -o game main.o graphics.o world.o
```

Rule: *target* is rebuilt if older than any $source_i$

... Sidetrack: Make/Makefiles

A `Makefile` for the example program:

```
game : main.o graphics.o world.o
        gcc -o game main.o graphics.o world.o

main.o : main.c world.h graphics.h
        gcc -Wall -Werror -std=c11 -c main.c

graphics.o : graphics.c world.h
        gcc -Wall -Werror -std=c11 -c graphics.c

world.o : world.c
        gcc -Wall -Werror -std=c11 -c world.c
```

Things to note:

- A *target* (`game`, `main.o`, …) is on a newline
  - followed by a `:`
  - then followed by the files that the target is dependent on
- The *action* (`gcc …`) is always on a newline
  - and must be indented with a *TAB*

... Sidetrack: Make/Makefiles

If `make` arguments are targets, build just those targets:

```
prompt$ make world.o
gcc -Wall -Werror -std=c11 -c world.c
```

If no args, build first target in the `Makefile`.

```
prompt$ make
gcc -Wall -Werror -std=c11 -c main.c
gcc -Wall -Werror -std=c11 -c graphics.c
gcc -Wall -Werror -std=c11 -c world.c
gcc -o game main.o graphics.o world.o
```

Exercise #10: Makefile

Write a `Makefile` for the binary conversion program (Exercise 6, Problem Set Week 1) and the new Stack ADT.

Summary

- Pointers
- Memory management
  - `malloc()`
    - aim: allocate some memory for a data object
      - the location of the memory block within heap is random
      - the initial contents of the memory block are random
    - if successful, returns a pointer to the start of the block
    - if insufficient space in heap, returns `NULL`
  - `free()`
    - releases a block of memory allocated by `malloc()`
    - argument must be the address of a previously dynamically allocated object
- Dynamic data structures

- Suggested reading:
  - pointers … Moffat, Ch. 6.6-6.7
  - dynamic structures … Moffat, Ch. 10.1-10.2
  - linked lists, stacks, queues … Sedgewick, Ch. 3.3-3.5, 4.4, 4.6
