

Chapter 19

Data Structures

Based on slides © McGraw-Hill
Additional material © 2004/2005 Lewis/Martin
Modified by Diana Palsetta

Data Structures

C allows a programmer to build a **type** that is a combination of more basic data type

The collection of basic data types is called as **data structure**

Data structure allows:

- Group related items together
- Organize them in memory so that its convenient to program and efficient to execute

Example

- An **array** is one kind of data structure (basically collection of integers)

CIT 593

19-2

Structures in C

keyword **struct** allows

- A mechanism for grouping related data of **different types**

Example

- Suppose we want to keep track of weather data for the past 100 days, and for each day, we want the following data

```
int highTemp;  
int lowTemp;  
double precip;  
double windSpeed;  
int windDirection;
```

We can use a **struct** to group these data

CIT 593

19-3

Declaring a struct

We name the **struct** and declare “fields” (or “members”)

```
struct w_type {  
    int highTemp;  
    int lowTemp;  
    double precip;  
    double windSpeed;  
    int windDirection;  
};
```

This is declaration so no memory is actually allocated yet!

CIT 593

19-4

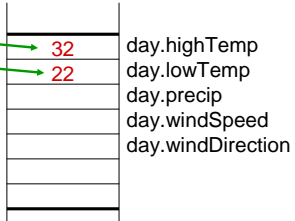
Defining and Using a struct

We define a variable using our new data type as follows:

```
struct w_type day;
```

Memory is now allocated (on stack), and we can access individual fields of this variable

```
day.highTemp = 32;  
day.lowTemp = 22;
```



32	day.highTemp
22	day.lowTemp
	day.precip
	day.windSpeed
	day.windDirection

Struct declaration allocates a contiguous memory for the collection

CIT 593

19-5

Declaring and Defining at Once

You can both declare and define a struct at same time

```
struct w_type {  
    int highTemp;  
    int lowTemp;  
    double precip;  
    double windSpeed;  
    int windDirection;  
} day;
```

Can still use the w_type name to declare other structs

```
struct w_type day2;
```

CIT 593

19-6

typedef

C provides a way to define new data type with meaningful names

Syntax

```
typedef <type> <name>;
```

Examples

```
typedef int Color; //makes color synonym for int  
typedef struct w_type WeatherData; //makes Weather  
synonym for struct w_type
```

Note: typedef provides no additional functionality, just there to give clarity to the code

CIT 593

19-7

Using typedef

Benefit

- Code more readable because we have application-specific types

```
Color pixels[128*124];  
WeatherData day1, day2;
```

Common practice

- Put typedef's into a header file
- Use type names in program
- If the definition of Color/WeatherData changes, *might* not need to change the code in rest of program

CIT 593

19-8

Memory allocation for struct on Runtime Stack

Consider the following code

```
...
int x;
WeatherData day;
int y;

day.highTemp = 12;
day.lowTemp = 1;
day.windDirection = 3;
...
```

offset = 0	y
1	day.highTemp
2	day.lowTemp
3	day.precip
4	day.windSpeed
5	day.windDirection
6	x

Note: $\text{addr}(x) = R5 + \text{offset} = R5 + 6$
R5 = Frame Pointer

CIT 593

19-9

LC3 Code for Structs

```
ADD R0, R5, #1 ; R0 has base addr of data structure day
; day.highTemp = 12;
AND R1, R1, #0
ADD R1, R1, #12 ; R1 = 12
STR R1, R0, #0 ; store value into day.highTemp

; day.lowTemp = 1;
AND R1, R1, #0
ADD R1, R1, #1 ; R1 = 1
STR R1, R0, #1 ; store value into day.lowTemp

; day.windDirection = 3;
AND R1, R1, #0
ADD R1, R1, #3 ; R1 = 3
STR R1, R0, #4 ; store value into day.windDirection
```

CIT 593

19-10

Array of Structs

Can define an array of **structs**

```
WeatherData days[100];
```

Each array element is a **struct**

To access member of particular element

```
days[34].highTemp = 97;
```

select element select field

CIT 593

19-11

Pointers to Struct

We can define and create a pointer to a **struct**

```
WeatherData *dayPtr; //create a ptr to point to weatherdata
dayPtr = &days[34];
```

To access a member of the **struct** addressed by **dayPtr**

```
(*dayPtr).highTemp = 97;
```

Dot (.) operator has higher precedence than *, so this is **NOT** the same as

```
*dayPtr.highTemp = 97; //so must put brackets around *dayPtr
Compiler will give an error for the above statement
```

Special syntax for this common access pattern

```
dayPtr->highTemp = 97;
```

CIT 593

19-12

Passing Structs as Arguments

Unlike an array, structs **passed by value**

- struct members copied to function's activation record, and changes inside function are not reflected in the calling routine's copy

Most of the time, you'll want to pass a **pointer** to a struct

```
Void printDay(WeatherData *day){
    ...
    printf("Low temp in deg F:\n" day->lowTemp);
    ...
}
void getInputDay(WeatherData *day)
{
    printf("High temp in deg F: ");
    scanf("%d", &day->highTemp);
}
} CIT 593
```

19-13

Dynamic Allocation

Problem

- What if we don't know the number of days for our weather program?
- Can't allocate array, because don't know maximum number of days that might be required
- Even if we do know the maximum number, it might be wasteful to allocate that much memory because most of the time only a few days' worth of data is needed

Solution

- Allocate storage dynamically, as needed

CIT 593

19-14

malloc

C Library (stdlib.h) function for allocating memory at run-time

```
void *malloc(int numBytes);
```

Returns

- **Generic pointer** (void*) to contiguous region of memory of requested size (in bytes)

Bytes are allocated from memory region called **heap**

- Heap != stack
- Run-time system keeps track of chunks of memory in heap that have been allocated

CIT 593

19-15

Purpose of the heap

If you write a function whose job is to create and populate a data structure

- suppose that data structure is placed on the stack then data structure **disappears** when you exit the function

You need dynamic storage that *you*, the programmer, can control

- the **heap** provides that dynamic storage

CIT 593

19-16

Using malloc

Problem

- Data type sizes are implementation specific

Solution

- Use `sizeof` operator to get size of particular type

```
malloc(n * sizeof(WeatherData));
```

Also need to change type of return value to proper kind of pointer

- Called "casting"

Finally to dynamically declare we do:

```
WeatherData *days;  
days = (WeatherData*)malloc(n* sizeof(WeatherData));
```

CIT 593

19-17

Example

```
int numberOfDays;  
WeatherData *days;  
  
printf("How many days of weather?");  
scanf("%d", &numberOfDays);  
  
days = (WeatherData*) malloc(sizeof(WeatherData)  
                             * numberOfDays);  
  
if (days == NULL) {  
    printf("Error in allocating the data array.\n");  
    ...  
}  
  
(days+1)->highTemp =  
OR  
days[1].highTemp = ...
```

If allocation fails, `malloc` returns NULL (zero)

Note: Can use pointer or array notation

CIT 593

19-18

free

When program is done with `malloced` data

- It must/should be released for reuse
- Achieved via `free` function
- `free` is passed same pointer returned by `malloc`

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

Example: `free(days)` //days is pointer WeatherData type

If allocated data is not freed

- Program may run out of heap memory and be unable to continue

Explicit memory management in C versus garbage collection in Java

CIT 593

19-19

Problems w/ explicit storage deallocation

There are two potential errors when deallocating (freeing) storage yourself (as in C):

- Deallocating too soon, so that you have *dangling references* (pointers to storage that has been freed and possibly reused for something else)
- Forgetting to deallocate, so that unused storage accumulates and you have no more heap memory left

CIT 593

19-20

Garbage Collection

The name "garbage collection" implies that objects that are no longer needed by the program are "garbage" and can be thrown away.

A *garbage collector* automatically searches out garbage and deallocates it

Practically every modern language, except C/C++, uses a garbage collector

Advantage: relieves programmers from the burden of freeing allocated memory because knowing when to explicitly free allocated memory can be very tricky

CIT 593

19-21

Garbage collection in Java

Java's garbage collector runs when it needs to, or when it wants to

- hopefully it runs "in the background"

It is reliable, but may cause unexpected slowdowns

You can ask Java to do garbage collection when the program has some time to spare

- but not all implementations respect your request

CIT 593

19-22