CS 135: Computer Architecture I

Instructor: Prof. Bhagi Narahari
Dept. of Computer Science
Course URL: www.seas.gwu.edu/~bhagiweb/cs135/

Next...

- Pointers and Arrays
  - Read Chapters 16, 18 of text

- Dynamic data structures
  - Allocating space during run-time
  - Read chapter 19 of text

Course Trivia...

- HW5, HW 6 posted
- Exam 2: Thursday Nov. 18th
  - All material from LC3 architecture to Implementation of C language

Course Trivia... After Exam 2

- Project 3
  - C programming project
- Project 4
  - Code performance optimization
  - Compiler optimizations
  - Due today
- Team homeworks 7,8
- Quiz 7, Quiz 8
- ...?
Passing by value is not enough

- Parameters are passed by value
- Arguments pushed onto run-time stack
- Consider the following function that's supposed to swap the values of its arguments.

```c
void Swap(int firstVal, int secondVal) {
    int tempVal = firstVal;
    firstVal = secondVal;
    secondVal = tempVal;
}
```

Executing the Swap Function

Before call:
- firstVal = valueA
- secondVal = valueB

After call:
- firstVal = valueB
- secondVal = valueA

These values changed...

...but these did not.

Swap needs addresses of variables outside its own activation record.
Address vs. Value

- Sometimes we want to deal with the address of a memory location, rather than the value it contains.

- Example:
  - Adding a column of numbers:
    - R2 contains address of first location.
    - Read value, add to sum, and increment R2 until all numbers have been processed.

- R2 is a pointer -- it contains the address of data we're interested in.

Pointers and Arrays

- We've seen examples of both of these in our LC-3 programs; now we'll see them in C.

- Pointer
  - Address of a variable in memory
  - Allows us to indirectly access variables
    - In other words, we can talk about its address rather than its value

- Array
  - A list of values arranged sequentially in memory
  - Example: a list of telephone numbers
  - Expression a[4] refers to the 5th element of the array a

Pointers in C

- C lets us talk about and manipulate pointers as variables and in expressions.

  - Declaration
    - int *p; /* p is a pointer to an int */
  - A pointer in C is always a pointer to a particular data type: int*, double*, char*, etc.

  - Operators
    - *p -- returns the value pointed to by p
    - &z -- returns the address of variable z

Pointers

- int i;
  - int *ip;

- Pointer: Variable that contains address of another variable.
- A pointer is a data object which is separate from what it points to.

- ip is a pointer to an integer
### Example

- `int i;`  
  - `int *ptr;`  
  - `*i = 4;`  
  - `ptr = &i;`  
  - `*ptr = *ptr + 1;`  

  **Store the value 4 into the memory location associated with i**  
  **Store the address of i into the memory location associated with ptr**  
  **Read the contents of memory at the address stored in ptr**  
  **Store the result into memory at the address stored in ptr**

### Example: LC-3 Code

```
/* i is 1st local (offset 0), ptr is 2nd (offset -1) */
*i = 4;
  AND R0, R0, #0 ; clear R0
  ADD R0, R0, #4 ; put 4 in R0
  STR R0, R5, #0 ; store in i
  *ptr = &i;
  ADD R0, R5, #0 ; R0 = R5 + 0 (addr of i)
  STR R0, R5, #1 ; store in ptr
  *ptr = *ptr + 1;
  LDR R0, R5, #-1 ; R0 = ptr
  LDR R1, R0, #0 ; load contents (*ptr)
  ADD R1, R1, #1 ; add one
  STR R1, R0, #0 ; store result where R0 points
```

### Pointers as Arguments

- Passing a pointer into a function allows the function to read/change memory outside its activation record.

```
void NewSwap(int *firstVal, int *secondVal)
{
  int tempVal = *firstVal;
  *firstVal = *secondVal;
  *secondVal = tempVal;
}
```

Arguments are integer pointers. Caller passes addresses of variables that it wants function to change.

### Swap

- A function that will swap two integers
- Last try:
  ```
  void swap(int *a, int *b)
  {
    int t;
    t = *a;
    *a = *b;
    *b = t;
  }
  ```
Now it works…

- We call it like this

```c
int x = 42;
int y = 84;
swap(&x, &y);
```

```c
void swap(int *a, int *b)
{
    int t;
    t = *a;
    *a = *b;
    *b = t;
}
```

Trace:

```c
int x = 42;
int y = 84;
swap(&x, &y);
```

```c
void swap(int *a, int *b)
{
    int t;
    t = *a;
    *a = *b;
    *b = t;
}
```

Trace:

```c
int x = 42;
int y = 84;
swap(&x, &y);
```

```c
void swap(int *a, int *b)
{
    int t;
    t = *a;
    *a = *b;
    *b = t;
}
```

Stack Frame for main:

```
x 42
y 84
```

Stack Frame for swap:

```
x 42
y 84
```
int x = 42;
int y = 84;
swap(&x, &y);

void swap(int *a, int *b) {
    int t;
    t = *a;
    *a = *b;
    *b = t;
}

void swap(int *a, int *b) {
    int t;
    t = *a;
    *a = *b;
    *b = t;
}

Passing Pointers

- How do you pass pointers in the activation record?
  - Using LC3 compiler...

- Parameters to the function are the addresses of the arguments!
### Passing Pointers to a Function

- `main()` wants to swap the values of `valueA` and `valueB`.
- It passes the addresses to `NewSwap`:  
  
  ```c
  NewSwap(&valueA, &valueB);
  ```

- **Code for passing arguments:**
  - `ADD R0, R5, #1 ; addr of valueB`
  - `ADD R6, R6, #0`
  - `ADD R0, R5, #0 ; addr of valueA`
  - `ADD R6, R6, #1 ; push`
  - `STR R0, R6, #0`

### Code Using Pointers

- Inside the `NewSwap` routine:
  ```c
  int tempVal = *firstVal;
  LDR R0, R5, #4 ; R0=xEFFA
  LDR R1, R0, #0 ; R1=M[xEFFA]=3
  STR R1, R5, #4 ; tempVal=3
  *firstVal = *secondVal;
  LDR R1, R5, #5 ; R1=EFF9
  LDR R2, R1, #0 ; R1=M[xEFF9]=4
  STR R2, R0, #0 ; M[xEFFA]=4
  *secondVal = tempVal;
  LDR R2, R5, #0 ; R2=3
  STR R2, R1, #0 ; M[xEFF9]=3
  ```

### Null Pointer

- Sometimes we want a pointer that points to nothing.
- In other words, we declare a pointer, but we’re not ready to actually point to something yet.

  ```c
  int *p;
  p = NULL; /* p is a null pointer */
  ```

- **`NULL`** is a predefined macro that contains a value that a non-null pointer should never hold.
  - Often, `NULL = 0`, because Address 0 is not a legal address for most programs on most platforms.

### Using Arguments for Results

- Pass address of variable where you want result stored:
  ```c
  scanf("%d ", &dataIn);
  ```
  - Useful for multiple results
  - Example:
    - return value via pointer
    - return status code as function result

- This solves the mystery of why ‘&’ with argument to `scanf`:
  - `scanf("%d ", &dataIn);`
  - Read a decimal integer and store in `dataIn`.
Address Operators

& Have a variable and want the address of it.

* Have address (or pointer) and want value of variable that it's pointing at.

Know this!

Arrays

• How do we allocate a group of memory locations?
  > character string
  > table of numbers

int num0;
int num1;
int num2;
int num3;

Array Syntax

• Declaration
  - type variable[num_elements];

  all array elements are of the same type
  number of elements must be known at compile-time

• Array Reference
  - variable[index];

  i-th element of array (starting with zero);
  no limit checking at compile-time or run-time

int ia[6];

• Allocates consecutive spaces for 6 integers
• How much space is allocated?
Arrays

- `int ia[6];`
- Allocates consecutive spaces for 6 integers
- How much space is allocated?
  - `6 * sizeof(int)`
- Also creates `ia` which is effectively a constant pointer to the first of the six integers
- What does `ia[4]` mean?

Arrays

- `int ia[6];`
- Allocates consecutive spaces for 6 integers
- How much space is allocated?
  - `6 * sizeof(int)`
- Also creates `ia` which is effectively a constant pointer to the first of the six integers
- What does `ia[4]` mean?
  - Multiply 4 by `sizeof(int)`. Add to `ia` and dereference yielding:

sizeof

- Compile time operator
- Two forms
  - `sizeof object`
  - `sizeof ( type name )`
- Returns the size of the object or the size of objects of type name in bytes
- Note: Parentheses can be used in the first form with no adverse effects

sizeof

- if `sizeof(int) == 4` then `sizeof(i) == 4`
- On a typical 32 bit machine...
  - `sizeof(*ip) -> 4`
  - `sizeof(ip) -> 4`
  - `char *cp;`  `sizeof(char) -> 1`
  - `sizeof(*cp) -> 1`
  - `sizeof(cp) -> 4`
  - `int ia[6];`
  - `sizeof(ia) -> 24`
Relationship between Arrays and Pointers

- An array name is essentially a pointer to the first element in the array

  ```c
  char word[10];
  char *cptr;
  cptr = word; /* points to word[0] */
  ```

- **Difference:**
  - Can change the contents of cptr, as in
  - `cptr = cptr + 1;`
  - *(The identifier "word" is not a variable.)*

Arrays

```c
int ia[6];
```

- `ia[4]` means *(ia + 4)*

Pointer Arithmetic

- Note on the previous slide when we added the literal 4 to a pointer it actually gets interpreted to mean
  - `4 * sizeof(thing being pointed at)`
  - This is why pointers have associated with them what they are pointing at!

Arrays

```c
int ia[6];
```

- Array elements are numbered like this since that’s how the pointer arithmetic works out!
Pointer Arithmetic

- Address calculations depend on size of elements
  - In our LC-3 code, we've been assuming one word per element.
    - e.g., to find 4th element, we add 4 to base address
  - It's ok, because we've only shown code for int and char, both of which take up one word.
  - If double, we'd have to add 8 to find address of 4th element.

- C does size calculations under the covers, depending on size of item being pointed to:
  - double x[10];
  - double *y = x;
    - *(y + 3) = 13;
    - allocates 20 words (2 per element)

  - same as x[3] = base address plus 6

Relationship between Arrays and Pointers

- An array name is essentially a pointer to the first element in the array
  - char word[10];
  - char *cptr;

    - cp = word; /* points to word[0] */

- Difference:
  - Can change the contents of cp, as in
  - cp = cp + 1;
  - (The identifier "word" is not a variable.)

Correspondence between Ptr and Array Notation

- Given the declarations, each line below gives three equivalent expressions:
  - *cptr word &word[0]
  - *(cptr + n) word + n &word[n]
  - *(cptr + n) *(word + n) word[n]

  - char word[10];
  - char *cptr;
  - cp = word; /* points to word[0] */

Passing Arrays as Arguments

- C passes arrays by reference
  - the address of the array (i.e., of the first element) is written to the function's activation record
  - otherwise, would have to copy each element

- main() {
  - int numbers[MAX_NUMS] =
    - #define MAX_NUMS 10
    - This must be a constant, e.g.
    - mean = Average(numbers);
  - ...
}
- int Average(int inputValues[MAX_NUMS]) {
  - ...
    - for (index = 0; index < MAX_NUMS; index++)
      - sum = sum + inputValues[index];
      - return (sum / MAX_NUMS);
  - }

A String Is an Array of Characters

- Allocate space for a string just like any other array:
  - char outputString[16];

- Space for string must contain room for terminating zero.

- Special syntax for initializing a string:
  - char outputString[16] = "Result = ";
  - ...which is the same as:
  - outputString[0] = 'R';
    outputString[1] = 'e';
    outputString[2] = 's';
    ...

I/O with Strings

- Printf and scanf use "%s" format character for string

- Printf -- print characters up to terminating zero
  - printf("%s", outputString);

- Scanf -- read characters until whitespace, store result in string, and terminate with zero
  - scanf("%s", inputString);

Array as a Local Variable

int foo(int myarray[])
{
    int grid[10];
    ...
}

Array as a Local Variable

- Array elements are allocated as part of the activation record.

- First element (grid[0]) is at lowest address of allocated space.

  If grid is first variable allocated, then R5 will point to grid[9].
**LC-3 Code for Array References**

- \( x = \text{grid}[3] + 1 \)
- \( \text{ADD R0, R5, } \#-9 \); \( R0 = \&\text{grid}[0] \)
- \( \text{LDR R1, R0, } \#3 \); \( R1 = \text{grid}[3] \)
- \( \text{ADD R1, R1, } \#1 \); \( \text{plus 1} \)
- \( \text{STR R1, R5, } \#-10 \); \( x = R1 \)
- \( \text{grid}[6] = 5 \)
- \( \text{AND R0, R0, } \#0 \)
- \( \text{ADD R0, R0, } \#5 \); \( R0 = 5 \)
- \( \text{ADD R1, R5, } \#-9 \); \( R1 = \&\text{grid}[0] \)
- \( \text{STR R0, R1, } \#6 \); \( \text{grid}[6] = R0 \)

**More LC-3 Code**

- \( \text{grid}[x+1] = \text{grid}[x] + 2 \)
- \( \text{LDR R0, R5, } \#-10 \); \( R0 = x \)
- \( \text{ADD R1, R5, } \#-9 \); \( R1 = \&\text{grid}[0] \)
- \( \text{ADD R1, R0, R1} \); \( R1 = \&\text{grid}[x] \)
- \( \text{STR R2, R1, } \#0 \); \( \text{grid}[x+1] = R2 \)

**Common Pitfalls with Arrays in C**

- Overrun array limits
  - There is no checking at run-time or compile-time to see whether reference is within array bounds.
  - \( \text{int array}[10]; \)
  - \( \text{for (i = 0; i <= 10; i++) array[i] = 0; } \)

- Declaration with variable size
  - Size of array must be known at compile time.
  - \( \text{void SomeFunction(int num_elements) \{ \}
  - \( \text{int tmp[num_elements]; \}
  - \( \}} \)

**Recall**

\[
\text{int ia[6];}
\]

\[
\text{ia[2] = 42;}
\]

Address calculation:
\[
2 \times \text{sizeof(}\ast \text{ia}) + \text{ia}
\]

Access is by dereferencing
\[
(2 \times \text{sizeof(}\ast \text{ia}) + \text{ia})
\]

Remember! You don’t type in the sizeof part!
What happens?

int ia[6];
ia[8] = 84;

Address calculation:
8 * sizeof(*ia) + ia

Remember! You don’t type in the sizeof part!

Stack Smashing

int another(int a, int b) {
  int x[4];

  x[0]
  x[1]
  x[2]
  x[3]
  x[4] Old FP
  x[5] Ret Addr
  x[6] Ret Val
  x[7] a
  x[8] b

  Old FP
  Return Addr
  Return Val
  a
  b

Multi-Dimensional Arrays
**Declaration**

```
int ia[3][4];
```

**Number of Rows**

**Number of Columns**

**Type**

**Address**

*Declaration at compile time*

*i.e. size must be known*

---

**How does a two dimensional array work?**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Column Major Order**

<table>
<thead>
<tr>
<th>0,0</th>
<th>0,1</th>
<th>0,2</th>
<th>0,3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
</tr>
<tr>
<td>0,2</td>
<td>1,1</td>
<td>2,0</td>
<td>2,1</td>
</tr>
<tr>
<td>0,3</td>
<td>1,2</td>
<td>2,1</td>
<td>2,2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Column 0</th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
</table>

**Row Major Order**

<table>
<thead>
<tr>
<th>0,0</th>
<th>0,1</th>
<th>0,2</th>
<th>0,3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
<td>0,2</td>
<td>1,0</td>
<td>1,1</td>
</tr>
<tr>
<td>0,2</td>
<td>0,3</td>
<td>1,1</td>
<td>1,2</td>
</tr>
<tr>
<td>0,3</td>
<td>1,2</td>
<td>1,3</td>
<td>1,3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Row 0</th>
<th>Row 1</th>
<th>Row 2</th>
</tr>
</thead>
</table>

---

**How would you store it?**

**Advantage**

- Using Row Major Order allows visualization as an array of arrays

```
ia[1]
```

<table>
<thead>
<tr>
<th>0,0</th>
<th>0,1</th>
<th>0,2</th>
<th>0,3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
<td>1,3</td>
</tr>
<tr>
<td>2,0</td>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
</tr>
</tbody>
</table>

```
ia[1][2]
```

<table>
<thead>
<tr>
<th>0,0</th>
<th>0,1</th>
<th>0,2</th>
<th>0,3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
<td>1,3</td>
</tr>
<tr>
<td>2,0</td>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
</tr>
</tbody>
</table>
Recall

- One Dimensional Array
  - int ia[6];
  - Address of beginning of array:
    - ia = &ia[0]

- Two Dimensional Array
  - int ia[3][6];
  - Address of beginning of array:
    - ia = &ia[0][0]
  - also
    - Address of row 0:
      - ia[0] = &ia[0][0]
    - Address of row 1:
      - ia[1] = &ia[1][0]
    - Address of row 2:
      - ia[2] = &ia[2][0]

Element Access

- Given a row and a column index
- How to calculate location?
- To skip over required number of rows:
  - row_index * sizeof(row)
    - row_index * Number_of_columns * sizeof(arr_type)
  - This plus address of array gives address of first element of desired row
  - Add column_index * sizeof(arr_type) to get actual desired element

Element Address =

Array_Address +
Row_Index * Num_Columns * Sizeof(Arr_Type) +
Column_Index * Sizeof(Arr_Type)

Element Address =

Array_Address +
(Row_Index * Num_Columns + Column_Index) *
Sizeof(Arr_Type)

What if array is stored in Column Major Order?

Element Address =

Array_Address +
(Column_Index * Num_Rows + Row_Index) *
Sizeof(Arr_Type)

0,0 0,1 0,2 0,3 1,0 1,1 1,2 1,3 2,0 2,1 2,2 2,3

0,0 1,0 2,0 0,1 1,1 2,1 0,2 1,2 2,2 0,3 1,3 2,3
How does C store arrays

- Row major
  - Pointer arithmetic stays unmodified
- Remember this....
  - Affects how well your program does when you access memory

Now think about

- A 3D array
  
  int a

- A 3D array
  
  int a[5]

- A 3D array
  
  int a[4][5]
Now think about

• A 3D array

\[
\text{int } a[3][4][5]
\]

Offset to \(a[i][j][k]\)?

• A 3D array

\[
\text{int } a[3][4][5] \\
[\text{slices}][\text{rows}][\text{columns}]
\]

\[
\text{offset} = (i \times \text{rows} \times \text{columns}) + (j \times \text{columns}) + k
\]

Structures

• Programs are solving a ‘real world’ problem
  
  ➢ Entities in the real world are real ‘objects’ that need to be represented using some data structure
    ➢ With specific attributes
  
  ➢ Objects may be a collection of basic data types
    ➢ In C we call this a structure

Data Structures

• A data structure is a particular organization of data in memory.
  
  ➢ We want to group related items together.
  
  ➢ We want to organize these data bundles in a way that is convenient to program and efficient to execute.

• An array is one kind of data structure.
  
  • struct — directly supported by C
  
  • linked list — built from struct and dynamic allocation
Structures in C

- A struct is a mechanism for grouping together related data items of different types.
  - Recall that an array groups items of a single type.

Example:
We want to represent an airborne aircraft:
- char flightNum[7];
- int altitude;
- int longitude;
- int latitude;
- int heading;
- double airSpeed;

We can use a struct to group these data together for each plane.

Defining a Struct

- We first need to define a new type for the compiler and tell it what our struct looks like.

```c
struct flightType {
    char flightNum[7]; /* max 6 characters */
    int altitude; /* in meters */
    int longitude; /* in tenths of degrees */
    int latitude; /* in tenths of degrees */
    int heading; /* in tenths of degrees */
    double airSpeed; /* in km/hr */
};
```

- This tells the compiler how big our struct is and how the different data items ("members") are laid out in memory.
- But it does not allocate any memory.

Declaring and Using a Struct

- To allocate memory for a struct, we declare a variable using our new data type.

```c
struct flightType plane;
```

- Memory is allocated, and we can access individual members of this variable:

```c
plane.flightNum[0]
plane.altitude = 10000;
```

- A struct’s members are laid out in the order specified by the definition.

Defining and Declaring at Once

- You can both define and declare a struct at the same time.

```c
struct flightType {
    char flightNum[7]; /* max 6 characters */
    int altitude; /* in meters */
    int longitude; /* in tenths of degrees */
    int latitude; /* in tenths of degrees */
    int heading; /* in tenths of degrees */
    double airSpeed; /* in km/hr */
} maverick;
```

- And you can use the flightType name to declare other structs.

```c
struct flightType iceMan;
```
typedef

- C provides a way to define a data type by giving a new name to a predefined type.
- Syntax:
  - typedef <type> <name>;
- Examples:
  - typedef int Color;
  - typedef struct flightType Flight;
  - typedef struct ab_type {
      int a;
      double b;
    } ABGroup;

Using typedef

- This gives us a way to make code more readable by giving application-specific names to types.
- Examples:
  - Color pixels[500];
  - Flight plane1, plane2;

Typical practice:
- Put typedef's into a header file, and use type names in main program. If the definition of Color/Flight changes, you might not need to change the code in your main program file.
  > Pay attention.....need this in your Project 3,4

Generating Code for Structs

- Suppose our program starts out like this:
  - int x;
  - Flight plane;
  - int y;
  - plane.altitude = 0;
  - ...

- LC-3 code for this assignment:
  - AND R1, R1, #0
  - ADD R0, R5, #-13 ; R0=plane
  - STR R1, R0, #7 ; 8th word

Array of Structs

- Can declare an array of structs:
  - Flight planes[100];

- Each array element is a struct
- To access member of a particular element:
  - planes[34].altitude = 10000;

- Because the [] and . operators are at the same precedence, and both associate left-to-right, this is the same as:
  - planes[34].altitude = 10000;
**Pointer to Struct**

- We can declare and create a pointer to a struct:
  - `Flight *planePtr;`
  - `planePtr = &planes[34];`
- To access a member of the struct addressed by `Ptr`:
  - `(*planePtr).altitude = 10000;`
  - Because the . operator has higher precedence than `*`, this is NOT the same as:
  - `*planePtr.altitude = 10000;`
- C provides special syntax for accessing a struct member through a pointer:
  - `planePtr->altitude = 10000;`

**Passing Structs as Arguments**

- Unlike an array, a struct is always passed by value into a function.
  - This means the struct members are copied to the function’s activation record, and changes inside the function are not reflected in the calling routine’s copy.
- Most of the time, you’ll want to pass a pointer to a struct.

```c
int Collide(Flight *planeA, Flight *planeB) {
    if (planeA->altitude == planeB->altitude) {
        ...
    } else {
        return 0;
    }
}
```

**Static vs. Dynamic Allocation**

- There are two different ways that multidimensional arrays could be implemented in C.
- Static: When you know the size at compile time
  - A Static implementation which is more efficient in terms of space and probably more efficient in terms of time.
- Dynamic: what if you don’t know the size at compile time?
  - More flexible in terms of run time definition but more complicated to understand and build
  - Dynamic data structures
- Need to allocate memory at run-time – `malloc`
  - Once you are done using this, then release this memory – `free`
- Next: Dynamic Memory Allocation

**Dynamic Allocation**

- Size of all of our data structures have been defined statically
  - `int myarray[100]` reserves 100 locations
- What if size is only known at run-time?
  - Guess max size and allocate statically?
    - `int myarray[max_size]`
- Dynamic allocation
  - Ask for space at run-time
  - Need run-time support – call system to do this allocation
  - Provide a library call in C for users
  - Where do you allocate this space – heap
Typical Arrangement

- Stack grows towards zero
- Heap grows towards xFFFF
- Can run out of space!

Dynamic Allocation

- Suppose we want our program to handle a variable number of planes — as many as the user wants to enter.
  - We can’t allocate an array, because we don’t know the maximum number of planes 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 planes’ worth of data is needed.

  • Solution: Allocate storage for data dynamically, as needed.

malloc

- The Standard C Library provides a function for allocating memory at run-time: malloc.
  - void *malloc(int numBytes);
  - It returns a generic pointer (void*) to a contiguous region of memory of the requested size (in bytes).
  - The bytes are allocated from a region in memory called the heap.
  - The run-time system keeps track of chunks of memory from the heap that have been allocated.

Using malloc

- To use malloc, we need to know how many bytes to allocate. The sizeof operator asks the compiler to calculate the size of a particular type.
  - planes = malloc(n * sizeof(Flight));
- We also need to change the type of the return value to the proper kind of pointer — this is called “casting.”
  - planes = (Flight*) malloc(n* sizeof(Flight));
```c
int airbornePlanes;
Flight *planes;

printf("How many planes are in the air?\n");
scanf("%d", &airbornePlanes);

planes = (Flight*) malloc(sizeof(Flight) * airbornePlanes);
if (planes == NULL) {
    printf("Error in allocating the data array.\n");
    ...
} else
    planes[0].altitude = ...

Note: Can use array notation or pointer notation.
```

Once the data is no longer needed, it should be released back into the heap for later use.

This is done using the `free` function, passing it the same address that was returned by `malloc`.

```c
void free(void*);
>
Free(planes[0]);
```

If allocated data is not freed, the program might run out of heap memory and be unable to continue.

Even though it is a local variable, and the values are 'destroyed', the allocator assumes the memory is still in use!

---

### Why use `malloc()`

- **Example: Linked list**
  - Read example in Chapter 19

- **You MUST get familiar with data structures and dynamic memory allocation**
  - Will need this for your project 3

---

### More on pointers, arrays..
Syntax for Pointer Operators

- Declaring a pointer
  - `type *var;
  
  Either of these work -- whitespace doesn't matter. Type of variable is `int` (integer pointer), `char` (char pointer), etc.

- Creating a pointer
  - `&var`
  - Must be applied to a memory object, such as a variable. In other words, `&3` is not allowed.

- Dereferencing
  Can be applied to any expression. All of these are legal:
  - `*var` contents of mem loc pointed to by `var`
  - `**var` contents of mem loc pointed to by memory location pointed to by `var`
  - `*3` contents of memory location 3

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>i</code></td>
<td></td>
</tr>
<tr>
<td><code>ip</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>i</code></td>
<td></td>
<td><code>int i;</code></td>
</tr>
<tr>
<td><code>ip</code></td>
<td></td>
<td><code>int *ip;</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>i</code></td>
<td>42</td>
<td><code>int i;</code></td>
</tr>
<tr>
<td><code>ip</code></td>
<td></td>
<td><code>int *ip;</code></td>
</tr>
<tr>
<td></td>
<td><code>i = 42;</code></td>
<td></td>
</tr>
</tbody>
</table>
### Pointers

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>42</td>
</tr>
<tr>
<td>ip</td>
<td></td>
</tr>
</tbody>
</table>

#### Code

```c
int i;
int *ip;
i = 42;
*ip = 84;
```

**ERROR!!!**
Core Dump if lucky

---

#### Translation

```c
int *pi = &i;
```
### Pointers

- **Name**
- **Contents**

<table>
<thead>
<tr>
<th>Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>???????????</td>
</tr>
<tr>
<td>ip</td>
<td></td>
</tr>
</tbody>
</table>

#### Code

```c
int i;
int *ip;
i = 42;
ip = &i;
*ip = &i;
```

#### NO!!!

- Powerful and dangerous
- No runtime checking (for efficiency)
- Bad reputation
- Java attempts to remove the features of pointers that cause many of the problems hence the decision to call them references
  - No address of operators
  - No dereferencing operator (always dereferencing)
  - No pointer arithmetic

### Question?

```c
int x = 3;
int y = 72;
int *px = &x;
int *py = &y;
*px = 7;
py = px;
x = 12;
printf("%d %d\n", *px, *py);
```

**What is the output?**

1) 3 72
2) 72 3
3) 7 12
4) 12 7
5) 3 3
6) 72 72
7) 12 12
8) 12 72
9) 72 12