APS105: Absolutely Official Pointers Factsheet

Daniel Zingaro

February 16, 2010

Instructions

Read once a day until April 2010. Then recycle.

Here is an outline of the most important facts and confusions about pointers. The exercises should be attempted as they are reached in the handout.

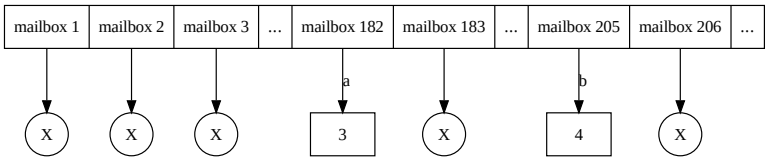
Fact 1: All Variables Have Addresses

You know when we say something like

```
int i;
i = 5;
```

? One important property of `i` is its value: it has value 5. But another important property is the variable's location, or address. The address is some value, like `25843` or `9122212`. We generally don't care what the address is, because as a raw address it is meaningless to us.

Think of memory as a bunch of mailboxes, each one having a different address. So, we have a mailbox for address 1, a mailbox for address 2, a mailbox for address 3, and so on. Each mailbox can hold one value. In the following diagram, mailbox (address) 182 is where `a` is stored, and mailbox 205 is where `b` is stored. These are the variables' addresses. If we were able to ask, "what is the address of `a`?", C would respond "182".



Fact 2: & Gives us the Address of a Variable

Let's say I declare a variable `a`:

```
int a = 3;
```

and I want to know the address of that variable. (In the above diagram, I want C to tell me "182".) How do I do this? I use the `&` operator, and say `&a`.

`&a` means "give me the address of `a`."

Even though an address looks like an integer (like 182), it's not a regular integer. It has special meaning as an address, and so we cannot say something like:

```
int myAddr = &a;
```

instead, we have special variables called pointers (stay with me here ...). A pointer variable is a variable that holds an address. Just like integer variables hold integers and float variables hold floats, pointer variables hold addresses of variables. To declare a pointer variable, we include a `*` in front of the variable's name:

```
int *p;
```

Read this as "p holds the address where I can find an integer". How can I initialize p? I require the address of an integer — and we know how to get one of those. We use & on an integer variable:

```
int a = 3;
int *p = &a;
```

If a is stored at address 182, then p will have the value 182. We say that p "points to" a. p is the address of where I can find an integer. What that means is that I can find an integer like this:

- 1. Get the value of p. It's 182.
- 2. Go to memory address 182, and get the value there. The value is 3.

The question is: how do I "go to address 182"?

Fact 3: * Tells us What is Stored at an Address

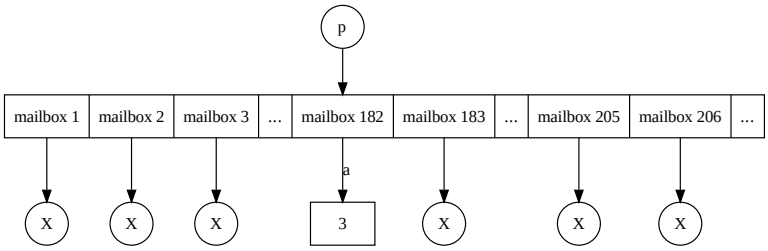
So, I have variable p, whose value is the address of where I can find an integer. Let's say the value of p is 182. I want to "follow the pointer" to address 182, and fetch the value stored there. I do this with the * operator.

*p means "get the address stored in p and tell me what is found at that address." (Or: "give me whatever is pointed to by p.")

So, *p means, "what is at address 182?"; the answer is 3. I can therefore print 3 in two ways, using variable a directly, and following a's address stored in p:

```
int a = 3;
int *p = &a;
printf ("%d\n", a);
printf ("%d\n", *p);
```

Here is the relationship between a and p.



Exercise 1

What value is printed by the following code? Be sure you understand the assignment c = b: it is copying the value of b into c. The value of b is the address of a.

```
int a = 3;
int *b;
int *c;
b = &a;
c = b;
printf ("%d\n", *c);
```

Fact 4: * can Store a Value in an Address

If we use `*` on the left-hand-side of an assignment statement, we can put a value in the address stored in a pointer variable. Here's an example.

```
int a = 3;
int *p = &a;
*p = 19;
```

Prior to the last line executing, you will agree that `a` and `*p` have the same value of 3. `a` and `*p` are therefore two names that refer to the same thing. When we say `*p = 19`, it is the same as `a = 19`. In other words, `*p = 19` stores 19 in `a`'s memory address. After this, we can print the value of `a` to see that it really has changed to 19. Try it!

Fact 5: Pointers let Functions Change Variables

Let's say we're in `main` and we want to call a function that changes our variable `i`:

```
int main(void) {
  int i = 18;
  /*call function to change value of i*/
  change (...);
  ...
}
```

We cannot pass `i` itself to a function that expects to change `main`'s `i`. If we try:

```
change (i);
```

then `change` is simply getting the value 18. `change` has no idea where the 18 came from; it does not know the address of that 18. It cannot possibly change the value stored in `i`.

Instead, let's pass the address of that 18:

```
change (&i);
```

Now, `change` is getting an address, and at that address it will find an integer. To define `change` properly, we must indicate that it is receiving the address where an integer can be found (i.e. a pointer to an integer). Here, we'll write `change` so that it sets the variable at the specified address to the value 42:

```
void change (int *p) {
  *p = 42;
}
```

Exercise 2

With this definition of `change`, fill in the call to `change` below so that `main`'s `i` gets value 42. Think carefully about what `change` is expecting you to pass: it wants the address of an integer, not an integer.

```
int main(void) {
  int i = 18;
  /*call function to change value of i*/
  change (...);
  ...
}
```

Recipe 1: Writing a Function to Change a Variable

To write a function that modifies a variable that you pass to it:

- 1. For each "outside" variable you want to modify in the function, put a `*` in front of its name in the function definition
  - `void change (int *p)`
- 2. For each such variable `p`, use `*p` when storing or retrieving the data at the address stored in `p`.

Exercise 3

Write a function that takes two parameters: `a`, a pointer to an integer, and `b`, an integer. Store the value of `b` at the address pointed to by `a`. Then, write a `main` function that properly calls your function, and demonstrates that it works.

Fact 6: All Variables Have a Type

If we say `int i;`, `i` can store integers. The type of `i` is `int`. Similarly, if we say `int *j;`, `j` can store addresses of integers (i.e. pointers to integers). The type of `j` is therefore `int *` or "pointer to `int`". The **only** values we can store in `j` are addresses of integers. In `j`, we **cannot** store integer values like `5`, float values like `8.2`, memory addresses of float values like `&f` (assuming `f` is a `float`), etc.

Confusion 1: Declaring Pointers

Both of these declarations are the same:

```
int *i;
int* i;
```

You'll see both in real C code. The position of the `*` does not matter: both declarations make `i` have type "pointer to `int`". However, the declarations tend to evoke different interpretations (that of course result in the same correct conclusion):

- `int *i` looks like we are defining `*i` as an `int`. If `*i` is an `int`, it means that when we follow the address stored in `i`, we get an `int`. The type of `i` must therefore be "pointer to `int`", or `int *`.
- `int* i` makes it look like we are defining `i` to have type `int *` (i.e. "pointer to `int`"). This is exactly the same interpretation as we arrived at for `int *i`.

Confusion 2: Should I use & or *??

- Use `&` when you have a variable and you want its address
- Use `*` when you have a pointer variable, and you want to access what it points to
- Use `*` in the first line of a function definition for each pointer parameter you want to receive
- Never use `&` in the first line of a function definition

Exercise 4

Here is a function that takes two pointers to integers, and returns the sum of the values that they point to.

```
int sum (int *a, int *b) {
  return (*a + *b);
```

- A. Why do we return `*a + *b` rather than `a + b`?
- B. Write a `main` function that declares two integer variables and then uses `sum` to calculate their sum. Print the sum with `printf` to test that everything is working properly.

Confusion 3: scanf and &

It's not always true that a `&` is required for a `scanf`. If we have a pointer already, we do not use `&`:

```
int a;
int *p = &a;
scanf ("%d", p);
```

If we said `&p` rather than `p`, we would be passing a pointer to a pointer to an integer. This is not what `scanf` is expecting: it is expecting a pointer to an integer, which is exactly what `p` on its own is.

Confusion 4: Initializing Pointers

We know that all variables must be initialized before used — if we do the following, we'll get garbage output:

```
int i;
printf ("Garbage: %d\n", i);
```

The situation is similar with pointers. If we just declare a pointer without first initializing it:

```
int *p;
```

then its value is going to be garbage. In other words, it contains some arbitrary address. If we try to store something there (like `*p = 1850;`), we're going to overwrite some arbitrary location in memory. It might make our program crash, or it might corrupt our data, or it might just seem to work (which is most dangerous of all!).

Confusion 5: Pointer Parameter or Return Value?

Let's say you want to write a function that gives you a random number between 1 and 6. There are two ways to do it:

- 1. Write a function that takes a pointer to an integer, and puts a random number in the variable pointed to by the pointer. The function does not return anything (e.g. `void` return).
- 2. Write a function that takes no parameters, and returns an integer.

For this particular function, the second choice is more natural: we can directly use it in a `printf` statement, and we can avoid pointers altogether. Here is a sample program that defines both of these functions and shows how they are called.

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

void func1 (int *i);
int func2 ();

void func1 (int *i) {
  *i = rand() % 6 + 1;
}

int func2 () {
  return (rand() % 6 + 1);
}

int main(void) {
  int randNum1, randNum2;
  srand ((unsigned)time (0));
  func1 (&randNum1);
  printf ("func1 gave me %d\n", randNum1);
  randNum2 = func2();
  printf ("func2 gave me %d\n", randNum2);
  printf ("I can also call func2 directly to get %d\n", func2());
  return 0;
}
```

For a function that swaps two values, or generates ten random numbers and puts them in ten different variables, we have no choice but to use pointers. A function can return only one value.

APS105: Absolutely Official Arrays Factsheet

Daniel Zingaro

March 3, 2010

Instructions

Read once a day until April 2010. Then keep in backpack for good luck.

Here is an outline of the most important facts and confusions about arrays. The exercises should be attempted as they are reached in the handout.

Confusion 1: Number of Array Elements

If I declare an array like this:

```
int a[5];
```

I am not allowed to do this:

```
a[5] = 8;
```

An array with five elements has elements at indices 0, 1, 2, 3, and 4. Trying to access `a[5]` would be trying to access the sixth element of the array — and we've only allocated five!

Fact 1: All Array Elements Have Addresses

Just like all regular variables have addresses, so does each element of an array. Consider this array declaration:

```
int a[5];
```

C finds a contiguous chunk of memory that's big enough for the entire array, and allocates each array element in order. If we assume that an integer is 4 bytes, C has to find a chunk of $5 * 4 = 20$ free bytes for the above array.

Let's say that C decides that array `a` is going to start at address 2293512. This will be the address of the first array element; that is, `&a[0]` will be 2293512. If we assume that integers take up four bytes, we can figure out the address of each of the other array elements by continuing to add 4 to this value. For example, `&a[1]` will be 2293516. We can see this in action if we write a program to print out the addresses of each of the array's elements:

```
#include <stdio.h>

int main (void) {
  int i;
  int a[5];
  for (i = 0; i < 5; i++)
    printf ("%d\n", &a[i]);
  return 0;
}
```

Running this on my computer right now, I get the following output. Your addresses will be different, but the important thing is that you understand how each address after the first one is found.

```
2293512
2293516
2293520
2293524
2293528
```

Exercise 1: Size of Array

How many bytes are occupied by the following array?

```
int dan[18];
```

Fact 2: Array Elements must be Initialized

If we declare an integer variable, we know that we'd better initialize its value first, before we use it. The following code is bad, because we are trying to add 5 to whatever arbitrary value is currently in `i`:

```
int i;
i = i + 5; // Bad!
```

The same thing holds for arrays. We cannot use an array value unless we've initialized it first.

```
int b[4];
printf ("First element: %d\n", b[0]; // Bad!
```

Exercise 2: Accessing Array Elements

Is the following program accessing any initialized data?

```
int b[4];
b[2] = 8;
printf ("%d\n", b[2]);
```

How about this one — is it accessing any uninitialized data?

```
int b[4];
b[2] = 8;
printf ("%d\n", b[3]);
```

Confusion 2: Capacity vs. Usage

Just because we declare an array of a specific size, it does not mean that the whole thing is always going to be used. Consider the following program that reads marks from the user. It stores a maximum of ten marks, or fewer if they type `-1` instead of a valid mark. It then attempts to print out all of the marks they entered.

```
#include <stdio.h>
#define MAX_MARKS 10

int main (void) {
  int nextMark, counter, i;
  int marks[MAX_MARKS];
  counter = 0;
  do {
    printf ("Enter a mark (0 to stop) ");
    scanf ("%d", &nextMark);
    if (nextMark != -1) {
      marks[counter] = nextMark;
      counter++;
    }
  } while (nextMark != -1 && counter < MAX_MARKS);
  printf ("OK! Here are all of the marks you entered.\n");
  for (i = 0; i < MAX_MARKS; i++) // ouch!
    printf ("%d ", marks[i]);
  printf ("\n");
  return 0;
}
```

If I run the program and enter less than ten marks, here's the mess I get:

```
Enter a mark (0 to stop) 83
Enter a mark (0 to stop) 74
Enter a mark (0 to stop) 90
Enter a mark (0 to stop) 58
Enter a mark (0 to stop) -1
OK! Here are all of the marks you entered.
83 74 90 58 -1 1996250044 4200240 2293544 4200342 4200240
```

The problem? Take a look at the for-loop at the end of the program. It assumes that the entire array is filled with valid data, but that might not be the case. Rather than assuming that `MAX_MARKS` is the "top" of the array, we have to use something else that takes into account the number of marks actually entered.

Exercise 3: Fix It!

Make a small change to the above for-loop so it prints only those marks that were actually entered.

Fact 3: Array Names are Addresses

Consider the following program.

```
#include <stdio.h>

int main (void) {
```

```
  int a[5];
  printf ("Address of first element: %d\n", &a[0]);
  printf ("Using array name: %d\n", a);
  return 0;
}
```

The output for me is as follows.

```
Address of first element: 2293516
Using array name: 2293516
```

Again, the actual address is unimportant. The important thing is the following.

The name of an array is a pointer to its first element.

So, `&a[0]` and `a` do the same thing: they both get you the address of an array's first element.

Fact 4: Pointers can Access Arrays

Check this out:

```
#include <stdio.h>

int main (void) {
  int a[5];
  a[0] = 8;
  printf ("Value of first element: %d\n", a[0]);
  printf ("Using array name: %d\n", *a);
  return 0;
}
```

`a` is a pointer to the first element. We know how to get whatever a pointer points at: use `*`. So, using `*a` gives us the value of the first array element.

By adding or subtracting integer `k` from a pointer that points to an element of an array, we can move to the right or left in that array. Here we go:

```
#include <stdio.h>

int main (void) {
  int a[5];
  a[0] = 8;
  a[1] = 123;
  printf ("Value of second element: %d\n", a[1]);
  printf ("Using array name: %d\n", *(a+1));
  return 0;
}
```

Exercise 4: Pointers Accessing Arrays

Why do we say `*(a + 1)` rather than `*a + 1`? What would `*a + 1` do?

Confusion 3: Pointers into Arrays

We can create a pointer that points to any element of an array. As far as that pointer is concerned, it thinks that it is pointing to the "base" of an array. It has no idea that it is pointing at any element besides the first one. Let's try:

```
#include <stdio.h>

int main (void) {
  int a[5];
  a[2] = 4321;
  int *p;
  p = a + 1; // p is pointing at a[1]!
  *p = 18; // change a[1]!
  printf ("Value of second element: %d\n", a[1]);
  printf ("Using array name: %d\n", *(a+1));
  printf ("Getting the value with p: %d\n", *p);
  printf ("Using p again: %d\n", p[0]);
  printf ("Getting a different element: %d\n", p[1]);
  return 0;
}
```

The last `printf` is quite interesting. Remember that `p` points to `a[1]`. Think of `p` as an alias for the array whose first value is `a[1]`. Thinking this way, `p[0]` is an alias for `a[1]`, and so `p[1]` must be an alias for `a[2]`.

Exercise 4: Moving Backwards

Use `p` here to set the value of `a[0]`.

APS105: Absolutely Official Strings Factsheet

Daniel Zingaro

March 8, 2010

Instructions

Read once a day until April 2010. Then wrap it up and give it to someone for their birthday.

Here is an outline of some important facts and confusions about strings. The exercises should be attempted as they are reached in the handout.

Fact 1: Strings Don't Really Exist

Hmmm?

Well, in other programming languages, there is a real `string` type. In C, there is not. What happens is in C is:

1. We use `char` arrays to hold characters of our strings
2. We terminate them with a `'\0'` character
3. All of C's string-processing functions rely on this `'\0'` terminator to detect when the string ends

Therefore, a "string" in C is just a `char` array that ends with a `'\0'`. Everything that is true of arrays is therefore true of strings, including:

- The first character is at index 0
- Elements of strings are not initialized until you give them values
- You should never try to access a character past the end of the string
- You can initialize strings when you declare them
- A string's characters are placed in order in memory, with the address of each character increasing by 1 each time we move to the right
- An array that will be used as a string is not necessarily using all of its elements. The string ends at the first `'\0'` character.

Confusion 1: Pointers into Strings

Remember: using an array name by itself gives us the address of its first element. Adding an integer `k` to a pointer makes it point `k` elements to the right. We can use both of these facts to access portions of strings. For example, let's say we declare:

```
char g[] = "hello";
```

(C will reserve space for six characters here.) I can print the entire string:

```
printf ("%s\n", g);
```

I can also print starting from the `'e'`:

```
printf ("%s\n", g + 1);
```

This works because `g + 1` is a pointer to the `'e'`. When we tell `printf` to output the string starting there, it finds `"ello"` before hitting the `'\0'` character.

Exercise 1

What would be printed if we used

```
printf ("%s\n", g + 5);
```

What would be printed if we used

```
printf ("%s\n", &g[1]);
```

In other words: starting `k` positions from the beginning of a string gives us another string (with `k` fewer characters).

Exercise 2

Consider the following code:

```
char s[] = "funny";
int i = strcmp (s + 2, "nny");
```

What is the value of `i`?

Fact 2: We can Implement Built-in String Functions

Many of C's built-in string functions do little more than process a character array that ends with `'\0'`. We can therefore gain considerable practice with strings by trying to implement them ourselves. Here is an implementation of `strlen`; it works by counting characters until the `'\0'` is found. We also include a `main` to demonstrate a call of this function:

```
#include <stdio.h>

int myStrlen (const char *s) {
  int i = 0;
  while (s[i] != '\0')
    i++;
  return i;
}

int main(void) {
  char a[] = "gimme";
  printf ("%d\n", myStrlen (a));
  return 0;
}
```

What if we wanted to implement `strcpy` using `strcat`? We could do it like this (again with a small `main` function for testing):

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

void myStrcpy (char *s1, const char *s2) {
  s1[0] = '\0';
  strcat (s1, s2);
```

```
}

int main (void) {
char first[20] = "once upon ";
char second[] = "a time";
myStrcpy (first, second);
printf ("First: %s\n", first);
printf ("Second: %s\n", second);
return 0;
}
```

Exercise 3

Here is a function that returns a pointer to a string's `'\0'` terminator. (In other words, it returns the address of the `'\0'` at the end of a string. It does not return `'\0'`; the return type is `char *`, not `char`.)

```
char *endOfString (const char *s) {
  while (*s != '\0')
    s++;
  return s;
}
```

Use this function to rewrite `myStrlen` in one line. (Hint: use pointer subtraction to determine how many elements come between the start of a string and its `'\0'` terminator.)

Exercise 4

Implement `strlen` by using a pointer to step through the characters of `s`, rather than by using an integer to access each element. Your loop will terminate when your pointer is pointing to the `'\0'`.

Confusion 2: Strings and Characters

A single character in single quotes is not a string. Similarly, a string cannot be treated like a character in single quotes. Consider this code:

```
char s[] = "tell";
```

We cannot try to print the `t` like this:

```
printf ("%c\n", s);
```

`s` here is not a character; it's a pointer to a character. The following, then, is legal:

```
printf ("%c\n", *s);
```

Example 1: Vowels in a String

Let's write a program that tells us the total number of vowels in a string. We'll use integer `i` to loop through each index in the string, and compare each character to the vowels:

```
#include <stdio.h>

int numVowels (const char *s) {
  int i = 0, total = 0;
  while (s[i] != '\0') {
    if (s[i] == 'a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'o'
     || s[i] == 'u')
      total++;
    i++;
  }
  return total;
}

int main (void) {
  printf ("%d\n", numVowels ("abcda"));
  return 0;
  }
```

Exercise 5

This solution involves a tedious expression that tests for each vowel. Rewrite the solution so that it uses a string constant initialized to the five characters `aeiou`. Then, to check whether each character in the string is a vowel, use `strchr`. (You should make only one call of `strchr` on each iteration.)

APS105: Absolutely Official Recursion Factsheet

Daniel Zingaro

March 15, 2010

Instructions

Read once a day until April 2010. Then read once a day until April 2010. ...

Here is an outline of some important facts and confusions about recursion. The exercises should be attempted as they are reached in the handout.

Fact 1: Always Include a Base Case

A base case for a recursive algorithm is a subproblem that we can solve without making another recursive call. The base case has to be so simple that we can look at it and say, "oh, I know the answer to this without doing anything".

Here is an example. Let's say we want a recursive function to find the sum of an array[1]. If I give the array `[2, 6, 1]`, I expect the answer 9.

What is the base case? Well, it's very easy to find the sum of a piece of an array with only one element. If we want the sum of `[6]`, we can immediately conclude that the answer is `6` without doing any extra work.

What about the recursive case? If we want the sum of array `[2, 6, 1]`, then we could find the sum of array `[6, 1]` and add 2 to that result. All of this leads to the following recursive algorithm:

```
#include <stdio.h>

int arraySum (int i, int n, int a[n]) {
  if (i == n-1)
    return a[i];
  return a[i] + arraySum (i + 1, n, a);
}

int main (void) {
  int a[] = {2, 4, 5};
  printf ("%d\n", arraySum (0, 3, a));
  return 0;
}
```

Hmmm. Perhaps the following version is more clear: it makes it explicit that we do the recursive call exactly when the base case does not hold.

```
int arraySum (int i, int n, int a[n]) {
  if (i == n-1)
    return a[i];
  else
    return a[i] + arraySum (i + 1, n, a);
```

[1] You can absolutely do this with a loop, and you probably should. Many recursive functions with only one recursive call can be written more simply as a loop.

Exercise 1

Why is the `else` not required in this function?

There is another potential base case for the array-sum example. To see it, think about the sum of zero array elements. The sum of no array elements is 0, so we can return 0 once we've processed the entire array. We again use `i` to keep track of our position in the array; when it reaches `n`, we have added up all elements of the array. This base case leads to:

```
int arraySum (int i, int n, int a[n]) {
  if (i == n)
    return 0;
  return a[i] + arraySum (i + 1, n, a);
}
```

Exercise 2

Rather than moving left to right through the array, write a version of `arraySum` that moves from right to left. You'll be able to eliminate the `i` parameter, since we can use the fact that the recursion terminates when `n` gets to value 0.

Fact 2: Recursive Case Makes Progress toward the Base Case

It's important that your recursive case brings you closer to the base case. For example, in the code above, notice how `i` gets closer and closer to `n` on each recursive call. This is good, because the base case occurs when `i` reaches `n`. If our recursive case decreased `i` rather than increasing it, we would get more and more distant from our base case (not good!).

Confusion 1: First in, First Out

When a function makes more than one recursive call, be sure that you properly think about calling and returning from the function. In particular, remember that recursion follows a first-in, first-out principle; in other words, recursive calls of a function must terminate before we can back up to earlier calls of the function. Every time we make a recursive call, we put the previous function on hold until the new call is completely finished.

Here is some code from lecture.

```
void recur (int i) {
  if (i == 0) {
    printf ("%d ", i);
    return;
  }
  for (int j = 0; j < 2; j++)
    recur (i - 1);
}
```

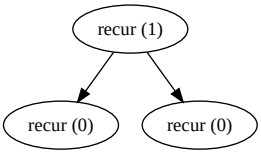
Here's what happens when I call the function with `recur (1)`:

1. `i` is initially 1. Let us refer to this call of the function as **call A**.
2. Since `i` is 1, the base case does not run, but the recursive case does
3. The recursive case consists of a loop that executes twice, for `j` having values 0 and 1.

4. The first time through the loop, we execute `recur (0)` (because `i - 1 = 1 - 1 = 0`). Let us refer to this function call as **call B**.
   - **Call B** starts with its `i` parameter having value 0. The base case therefore runs, printing `0`, and returning back to whoever made this function call.
5. We're now back in **Call A**, having just finished calling `recur (0)` the first time through the loop. Therefore, the next thing that **Call A** does is call `recur (0)` again, because of the second iteration of the loop. Let's refer to this next call of `recur (0)` as **Call C**.
   - **Call C** does the same thing as **Call B**, ultimately printing `0` and returning back to **Call A**.
6. But now, **Call A** has nothing left to do. It has already gone through its `j` loop twice, so it now terminates. Since **Call A** was the first call of the function, there is no further recursion to undo, and so our trace is complete.

Notice how we completely executed the first `recur (0)` before returning back to the call of `recur (1)` and executing the second `recur (0)`.

We can picture this recursive process as a tree, as follows.



Exercise 3

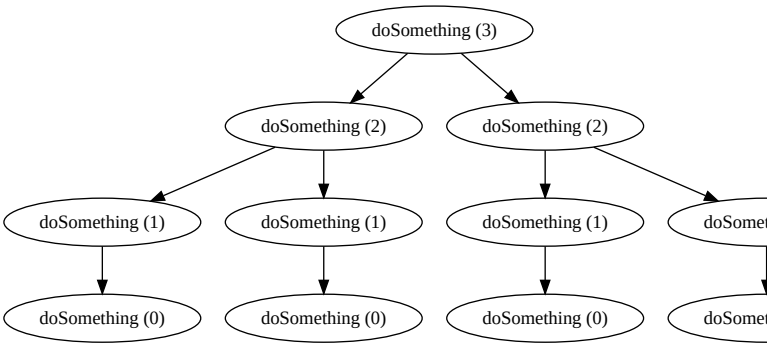
What is the output if we initially call `recur` as `recur (2)`? It might be helpful to draw the tree of recursive calls. There will be more levels in the tree than in the example above!

Exercise 4

Here is a recursive function.

```
void doSomething (int i) {
  if (i == 0)
    return;
  doSomething (i-1);
  printf ("%d\n", i);
  doSomething (i-1);
}
```

What happens when called as `doSomething (3)`? I provide the tree of calls below; make sure you can trace the function through each call and determine its output! Keep in mind that each function call is making two recursive calls, but it is doing a `printf` between those two calls. So, trace the first call to its completion, determine what the `printf` does, and then trace the second call to its completion — do this on each function call.



Observe that when we call `recur (3)`, we're calling `recur (2)`, printing the value `3`, and then calling `recur (2)` again. You therefore know that the output of `recur (3)` will be the result of `recur (2)`, the number 3, and then another copy of whatever `recur (2)` prints. You can apply this thinking on each recursive level as well. For example, `recur (2)`'s output is the output from `recur (1)`, the number 2, and then that same output from `recur (1)`.

Example 1: Generating All Permutations

Given `n` distinct letters, we can arrange them in `n!` ways. For example, if we have three distinct letters A B C—, we can arrange them in `3*2*1 = 6` ways. Those ways are: ABC, ACB, BAC, BCA, CAB, CBA.

Write a recursive function that prints all possible permutations of `n` distinct letters. The function will take an array of `n` elements, and print the contents of the array each time it represents a permutation.

Looking again at the permutations of A B C, we know that there are three ways to choose the first letter. Therefore, there are three groups of permutations that we must generate:

- Those that start with A
- Those that start with B
- Those that start with C

To generate the permutations that start with A, we move A to the beginning of our list of characters, and then generate all permutations of B C. This is a smaller version of our original problem, which we can solve recursively.

Once we have found all permutations that start with A, we want to find all those that start with B. We proceed in the same way as above: we move B to the beginning of the list, and recursively generate all permutations of A C. We do similarly to generate all permutations that start with C.

On each recursive call, our function must know which characters must be left alone, and which characters must be permuted. For example, when finding permutations that start with `A`, we want to leave `A` at the beginning and permute only `B C`. We use parameter `k` to indicate that character `k` is the first character that we can swap in order to find permutations.

Here is an outline of the algorithm:

- If `k` has value `n`, we have found a permutation. Print the contents of the array.
- Otherwise, for each character at position `k, k+1, ..., n`
  - Swap that character with the character at position `k`
  - Generate all permutations of characters at position `k+1, k + 2, ... , n`

The base case here occurs when `k` has value `n - 1`. This corresponds to the situation where the only characters we can permute are those with index at least `n - 1`. Since this corresponds to a single character, there is only one way to permute that character. When this happens, we have generated a complete permutation, so we can print the contents of the array.

One last complication. When `k` has some value `c`, we will make a recursive call with `k` having value `c + 1`. we cannot allow this `k + 1` call to modify the array that we see when it has finished. If we recursively call the function with array `['A', 'B', 'C']`, we must see the array as `['a', 'b', 'c']` when we regain control. This will not automatically happen: recall that arrays are passed as pointers, so they can be modified by function calls. We must make a copy of the array and pass **that** on each recursive call we make.

Exercise 5

Based on the above description, implement the function

```
void permutations (int n, int a[n], int k)
```

APS105: Quicksort

Daniel Zingaro

March 28, 2010

1 Why Quicksort?

We have covered three sorts to this point: selection, insertion, and bubble. They have a few things in common:

- They are all iterative. They use loops rather than recursion.
- They are all similar in terms of efficiency. That is, one of them is not conclusively better than the others. (But as we have seen in class, there are some special cases where one sort does substantially better than the others; for example, insertion sort on an already-sorted list.)
- They are all extremely slow. If you tried sorting millions of elements using these sorting algorithms, it would take forever. They are OK for small amounts of data, but degenerate quickly once a certain threshold is reached.

By contrast, quicksort is a recursive sorting technique that is extremely fast. It's more complicated to implement, and a little easier to get it wrong, but it's probably worth it if you're sorting a lot of data.

2 Idea Behind Quicksort

We can demonstrate the powerful idea of quicksort with an example. Let's say that we're sorting the following list of numbers:

```
8 9 1 6 4 2 3
```

Assume that we choose the value 3, and reorder the array so that all values less than 3 come first, followed by all values that are at least 3. There are many ways to rearrange the array; the most important thing is that values less than 3 are all found before any values that are at least 3. Here is one way to rearrange the array so that this is satisfied:

```
1 3 2 8 9 6 4
```

Rearranging the array in this way is referred to as **partitioning** the array.

Exercise 1: Another Rearrangement

Give another valid way of partitioning the array, again choosing value 3. That is, put all values less than 3 first, then put the values that are at least 3. There are lots of ways to do this!

Now, it is helpful to visualize our array as consisting of two separate pieces:

```
1 3 2 ::: 8 9 6 4
```

The essence of quicksort can be captured by the following idea.

If we can independently sort the left piece of the array, and then sort the right piece of the array, the entire array would be sorted.

That is, we can first sort:
`1 3 2`
and then sort:
`8 9 6 4`
at which point the entire array will be sorted.

How should we sort each of these two pieces? Keep in mind that what we have done here is take one big sorting problem (with seven elements) and produce two smaller, simpler sorting problems (one with three elements, one with four). We can therefore apply exactly the same idea to each subproblem. That is, to sort the left half:
`1 3 2`
we could partition this piece of the array around some value, to yield two further subproblems, such as: `1 2 ::: 3`
Eventually, our subproblems (such as the one-element array `3` here) will be so simple that we can solve them directly, without breaking them down any further. Once we have solved all subproblems for the left piece of the array, we proceed to break down: `8 9 6 4` into smaller sorting problems, until they too are so simple that further partitioning is not required.

It is important to understand that sorting the left half of the array, and sorting the right half of the array, are totally independent. Why? The reason is that when we partition an array, we have split the array into small values and large values. None of the small values can possibly go where any of the large values are, and vice versa. If we independently sort these two sides of the array and then think about pasting them together, the whole array will come out sorted.

Think about this in terms of sorting people. Choose some height, like 5 ft 5 in, put all people less than this height in one group, and all people more than this height in another group. Now, have the two groups put themselves in order based on height. You'll get two rows of people and, if you join the two rows, everyone will be ordered by height.

3 Partitioning an Array

We must be able to partition an array around some value `pivot`, so that all values less than `pivot` come first, followed by all other values. There are many different approaches for doing this. (For example, in the first exercise of this handout, you used whatever strategy you liked to partition the array.) I will expect you to be familiar with the method below (not the one in the textbook).

(The word **pivot** simply refers to the value that partitions the array. The usual terminology in quicksort discussions is to refer to this as the pivot value. For example, if we want all elements less than 9 to come first and then everything else, we say that 9 is the pivot value.)

The algorithm for partitioning an array processes the array from left to right. It uses two indices to remember the boundary between elements that are less than the pivot, those that are greater than or equal to the pivot, and those elements that have not yet been processed.

- We'll use variable `i` to keep track of the border between elements that are `< pivot` and those that are `>= pivot`
- We'll use variable `j` to keep track of the start of the elements we have not yet processed

`< i`	between `i` and `j`	`>= j`
`< pivot`	`>= pivot`	unprocessed

Here is the algorithm for partitioning an array `a` around a pivot:

- We begin with `i` and `j` at the left end of the part of the array we are sorting
- Then, at each step, we have two possibilities
  - If `a[j] >= pivot`, we simply increment `j`
  - Otherwise, `a[j] < pivot`, so we cannot just increment `j`
    - We swap `a[i]` and `a[j]`, and then
    - Increment both `i` and `j`

4 Partition Code

To complement my pseudocode description, here is code that implements the partition procedure. This function partitions the array between indices `low` and `high`, and returns the index of the first element that is `>= pivot`.

```
int partition (int low, int high, int pivot, int a[]) {
  int i = low, j = low;
  while (j <= high) {
    if (a[j] < pivot) {
      swap (a, i, j);
      i++;
    }
    j++;
  }
  return i;
}
```

5 Partition Example

Here is a worked example of how the above partition algorithm would operate on an array. Please follow along with the code above and/or its preceding description.

- Consider array: $[12, 30, 25, 8, 4, 9, 15, 13]$
- Let's partition around pivot 13
- Initial state: `i = 0`, `j = 0`
- Is `a[0] < pivot`? Yes — "swap" `a[0]` with itself and increment `i` and `j`
- New state: `i = 1`, `j = 1`
- Is `a[1] < pivot`? No — increment `j`
- New state: `i = 1`, `j = 2`
- Is `a[2] < pivot`? No — increment `j`
- New state: `i = 1`, `j = 3`
- Is `a[3] < pivot`? Yes — perform swap; increment both
- New array: $[12, 8, 25, 30, 4, 9, 15, 13]$
- New state: `i = 2`, `j = 4`
- Is `a[4] < pivot`? Yes — perform swap; increment both
- New array: $[12, 8, 4, 30, 25, 9, 15, 13]$
- New state: `i = 3`, `j = 5`

- Is `a[5] < pivot`? Yes — perform swap; increment both
- New array: $[12, 8, 4, 9, 25, 30, 15, 13]$
- New state: `i = 4`, `j = 6`
- Is `a[6] < pivot`? No — increment `j`
- New state: `i = 4`, `j = 7`
- Is `a[7] < pivot`? No — increment `j`
- New state: `i = 4`, `j = 8`

So, the final array, after partitioning around pivot value `13` is:
$[12, 8, 4, 9, 25, 30, 15, 13]$
All values less than 13 come first, followed by all values that are at least 13. We are going to come back to this shortly and show how to use quicksort on both of these pieces so that the entire array is sorted.

Exercise 2

Partition the array $[12, 8, 4, 9]$ around pivot value 9.

Partition the array $[12, 8, 4, 9]$ around pivot value 5. (Don't get confused because 5 is not in the array. Continue to use the partition algorithm to put values less than 5 before any values that are greater than or equal to 5.)

6 Complete Quicksort Algorithm

Now, we have a way of partitioning an array so that all values less than the pivot are placed before any value that is $\geq$ the pivot. We use this to write the quicksort algorithm, as follows:

- If we are asked to sort a piece of an array with zero or one elements, we do nothing. An empty array, or an array with one element, is already sorted. This is the base case.
- Otherwise, we are sorting a piece of an array that has at least two elements. In this case:
  - (1) Partition the array around a pivot value to divide the array into those values that are less than the chosen pivot, and those values that are at least as large as the pivot
  - (2) Swap the pivot value so that it sits between both of these sections of the array
  - (3) Recursively call quicksort on the values that are smaller than the pivot, and
  - (4) Recursively call quicksort on the values that are at least as large as the pivot

The chosen pivot can be any value; the code below chooses the rightmost value on each recursive call.

```
void quicksort (int low, int high, int a[]) {
  if (low < high) {
    int pivot = a[high];
    int i = partition (low, high - 1, pivot, a);
    swap (a, i, high);
    quicksort (low, i - 1, a);
    quicksort (i + 1, high, a);
  }
}
```

Exercise 3

The choice of pivot ends up being critical to the efficiency of quicksort. If we choose a pivot that evenly divides the array into two halves, we are happy. If instead we choose a

pivot that divides the array into one huge piece and one small piece, we have not made as much progress. Assuming we use the rightmost element as the pivot value, give an example of an array that will result in a poor pivot being chosen on each recursive call, and hence poor performance for quicksort.

7 Continuing the Example

7.1 Sorting the Entire Array

Let's go back to our partitioned array in section 5. We pivoted around value 13, and ended up with:
$[12, 8, 4, 9, 25, 30, 15, 13]$
To continue the sort, we will now use the ideas of section 6.

Our original array contained more than one element, which is why we partitioned the array around a pivot in the first place. In other words, we have just completed step (1) of the recursive case for quicksort, and now must complete steps (2), (3) and (4) in order to sort the entire array.

Step (2) — swapping the pivot in between those elements that are less and those elements that are at least as large – gives:
$[12, 8, 4, 9, 13, 30, 15, 25]$

Step (3) says that we should now call quicksort on the subarray:
$[12, 8, 4, 9]$
In other words, we stop what we're doing on the entire array, and start the quicksort algorithm from the beginning on this smaller array. When we're finished with this smaller subarray, we'll come back here and do step (4) on the whole array, at which point we'll be finished.

7.2 Sorting $[12, 8, 4, 9]$

We now run quicksort on:
$[12, 8, 4, 9]$
Look back at section 6. The array contains more than one element, so we are not done; instead, we carry out the four steps of the recursive case: (1) partition, (2) swap pivot into the middle, (3) quicksort small values, (4) quicksort large values.

(1) We choose 9 as the element around which to pivot this subarray. Running the partition algorithm from section 5, we get:
$[4, 8, 12, 9]$

(2) Swapping the pivot between the small and large elements gives:
$[4, 8, 9, 12]$

(3) We make a recursive call of quicksort on the subarray:
$[4, 8]$
which causes us to suspend our sorting of $[4, 8, 9, 12]$ right here, sort $[4, 8]$, and then come back and carry out step (4) to sort the elements to the right of 9.

7.3 Sorting $[4, 8]$

Does this subarray contain more than one element? Yes; so we must execute the recursive case of quicksort. Again, four steps:

(1) Partition around pivot element 8. This causes no change:
$[4, 8]$
(2) Swap the pivot into place. Again, no change:
$[4, 8]$
(3) Recursively call quicksort on those elements smaller than the pivot. That is, we call quicksort recursively on:
$[4]$

7.4 Sorting $[4]$

We have reached a base case: an array with one element. We do nothing.

7.5 Returning to Sorting $[4, 8]$

At this level, we have just carried out step (3). We must now carry out step (4) — sorting all elements to the right of 8. There are no such elements, so when we make this recursive call, quicksort will immediately return (the situation is the same as when we asked quicksort to sort [4]).

7.6 Returning to Sorting $[12, 8, 4, 9]$

At this level, we have just finished step (3), with the following array:
$[4, 8, 9, 12]$
We must now do step (4). Recall that our pivot on this level is 9, and step (4) involves quicksorting those elements to the right of 9. We will therefore call quicksort recursively on subarray [12]. Since this subarray contains only one element, quicksort will return immediately.

7.7 Back to the Top

Remember our original partition, where we pivoted around value 13? Well, we have now finished steps (1), (2) and (3) on that array. We can now continue with step (4) on this entire array, picking up from where we were interrupted at the end of section 7.1.

Our current array is:
$[4, 8, 9, 12, 13, 30, 15, 25]$
and we are going to call quicksort recursively on:
$[30, 15, 25]$
The pattern is hopefully clear. We are not asking to sort an array of less than two elements, so the recursive case of quicksort will be evoked. We will therefore carry out the four steps of the recursive case: pivot around 25, swap 25 in between both sections, sort values < 25, sort values ≥ 25 ...

Exercise 4

Finish the quicksort!

APS105: Dynamic Memory Allocation

Daniel Zingaro

March 29, 2010

1 Why Dynamic Memory Allocation?

Up to this point in the course, C has automatically managed our programs' memory for us. For example:

- (1) When a function is called, memory for the function's local variables and parameters is allocated. When the function returns, that memory is automatically released. We can call a function as many times as we like and not worry about how that memory gets acquired and freed.
- (2) When we declare an array with a specific number of elements, C automatically allocates a chunk of memory in which to store these elements. We are assured that when we start writing into the array's elements, we will be writing into valid memory that has been properly allocated to our program. This is also true for structures: C handles the details of allocating memory for our structure variables.

Both of these facts greatly simplify writing programs. It would be a chore to have to manually allocate space for each variable when we call a new function, or to have to explicitly state how much memory is required for an array of a certain size. However, both of these C features also restrict what we can do with the memory our program uses. For example:

- (Drawback of (1)) What if we want to allocate some memory in a function, but we do not want to lose that memory when the function terminates? That is, we want to be able to put some value into a local variable, and make that variable available long after the function returns. For example, what if we wanted to write a function that put a random phrase into a newly allocated string, and returned a pointer to that string? We cannot let the standard behavior of local variables (losing their memory when the function terminates) take place; we must take greater control of memory management.
- (Drawback of (2)) What if we do not know how big to make an array? Arrays have the property that once we choose their size, we're stuck with that choice until the function terminates. We might assume, for example, that 50 elements will be enough for our purposes. But then, the user might type in way more stuff than we expect, so the array will not be able to hold it all. If we let C manage the array's memory, we must stop accepting data, since we have nowhere to put it (we cannot extend the array). But if we take control of memory management, we can change the size of the array as required. We can increase its size to accommodate more data, and even take away array elements that are no longer being used.

2 Malloc

To take control of memory management, there are two important functions.

- `malloc`: we tell this function how many bytes of memory we require, and it finds a chunk of memory that can satisfy this request. It then returns a pointer to the beginning of this chunk of memory. We can use this chunk of memory for whatever we

like. It is guaranteed to be contiguous, so we can use it to store an array, a structure, a string, a single variable, and so on.

- `free`: releases the memory we obtained using `malloc`. We use `free` when we're finished using a piece of memory.

Let's begin by studying the following program, which does not use `malloc` at all.

```
#include <stdio.h>

int main(void) {
  int howMany;
  printf ("How many grades? ");
  scanf ("%d", &howMany);
  int grades[howMany];
  printf ("Enter all grades.\n");
  for (int i = 0; i < howMany; i++)
    scanf ("%d", &grades[i]);
  printf ("\n***You entered***\n");
  for (int i = 0; i < howMany; i++)
    printf ("%d\n", grades[i]);
  return 0;
}
```

This program begins by asking the user how many marks they wish to enter. It then declares an array to hold the specified number of marks, and then asks for and stores each grade in the array.

We could use `malloc` to accomplish the very same thing, as shown in the next program.

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

int main(void) {
  int howMany;
  printf ("How many grades? ");
  scanf ("%d", &howMany);
  int *grades;
   grades = malloc (howMany * sizeof(int));
  printf ("Enter all grades.\n");
  for (int i = 0; i < howMany; i++)
    scanf ("%d", &grades[i]);
  printf ("\n***You entered***\n");
  for (int i = 0; i < howMany; i++)
    printf ("%d\n", grades[i]);
  return 0;
}
```

- We have added the `stdlib.h` include file at the top of the program. This is required if we want to call `malloc`. (This is similar to the `stdio.h` requirement for when we want to use `printf`.)
- We have replaced the array with a pointer `grades`. We then give `grades` the return value from `malloc`, which is a pointer to a new block of memory large enough to store the array's contents. We want to be able to store `howMany` integers, and the size of each

of these integers is `sizeof (int)`. Therefore, we multiply these two values in order to find the total number of bytes required.

What we are doing here is simulating exactly what C would have done for an array. We have asked for the appropriate number of bytes, so that we can use that block of memory just like an array.

Today, we typically do not use `malloc` to allocate memory for an array (as we did here). However, in the early days of C, there was a restriction that goes like this: "the number of elements of an array must be determined by a constant value". What that meant was that we had to decide on an array's size before we compiled the program. We could choose a value like 5, or 10, or 20, but no matter what, the constant value itself had to be inside the `.c` file and could not be the value of a variable at runtime. This was quite problematic. If we wrote a program to manage the marks of students, we would have to guess before compiling the program how many students we would be able to handle. 30? Well, what if someone wanted to use the program for a huge class. 1000? Now we will waste tons of memory for small classes! The resolution, of course, was to use `malloc`, allocating memory once we have determined exactly how much to allocate.

But, as said, today we don't have to worry about doing that, so our `malloc` example above, in the context of storing grades, is more educational than practical.

3 Free

Once we use `malloc`, C no longer manages that piece of memory for us. Specifically, we are required to free (release, give up) the memory once we are done with it. To do so, we use the `free` function. We can add a call of `free` to our prior example, as follows.

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

int main(void) {
  int howMany;
  printf ("How many grades? ");
  scanf ("%d", &howMany);
  int *grades;
   grades = malloc (howMany * sizeof(int));
  printf ("Enter all grades.\n");
  for (int i = 0; i < howMany; i++)
    scanf ("%d", &grades[i]);
  printf ("\n***You entered***\n");
  for (int i = 0; i < howMany; i++)
    printf ("%d\n", grades[i]);
  free (grades);
  return 0;
}
```

The truth is that memory that our program uses is automatically freed when the program terminates, so in a practical sense this call of `free` does nothing. But it is very good style to always `free` everything that was allocated with `malloc`.

4 Example of malloc and free

Here is a real example of a function that uses `malloc`, and a `main` function that uses `free` to release memory when it is no longer required. The function `getJoke` uses a **local**

variable array to store the beginning of one of two bad jokes. We want to return this joke from the function, so that the caller can print it. Unfortunately, `choice` is a local variable, whose memory is destroyed when the function returns. If we were to say `return choice;`, our caller would get a pointer to the first character of the joke. But, the characters of the joke are going to be long gone, because the function has already terminated.

Instead, what we do is allocate our own memory using `malloc`. We know that **this** memory will not go away when the function finishes. When we `return newString`, we're giving our caller a pointer to the first character of the joke, and our caller can safely use this to read the entire string starting from that first character.

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

char *getJoke () {
  char choice[100];
  int i = rand () % 2; /*Two jokes*/
  if (i == 0)
    strcpy (choice, "Why did the chicken...");
  else if (i == 1)
    strcpy (choice, "What did one wall...");
  /*Our joke is now in choice. But choice is a local variable!
  It is destroyed when the function returns.
  To get permanent memory, we use malloc.*/
  char *newString = malloc (strlen (choice) + 1);
  strcpy (newString, choice);
  return newString;
}

int main (void) {
  srand (time(0));
  char *j;
  j = getJoke ();
  printf ("%s\n", j);
  /*Done with the first joke...*/
  free (j);
  j = getJoke (); /*Get another joke*/
  printf ("%s\n", j);
  free (j);
  return 0;
}
```

Two points about the `malloc` call:

- We ask for one more byte than `strlen (choice)`; this extra byte is going to be used to store the `'\0'` character that gets copied by `strcpy`.
- There's no `sizeof` expression in this `malloc` call. The reason is that we're assuming characters to be one byte, so `sizeof (char)` will be 1.

I'd like you to think back to lab 4, where you wrote a `subtract` function that placed characters in the `result` string. We assumed that `result` had enough space in it for the

characters; it was the caller's responsibility to declare an array that could be passed for the `result` parameter. You now know that we could have used `malloc` as an alternative. Rather than having `result` as a parameter and having the caller of the function declare an array with enough space, we could have returned a pointer to the result, allocating enough memory inside the function using `malloc`.

5 Declaring and Allocating

Let's say we wanted to use `malloc` to allocate enough memory to store one integer. We could do it like this:

```
int *p; /*Pointer to int (pointing to random memory)*/
p = malloc (sizeof (int));
*p = 42; /*Store 42 in the memory we got from malloc*/
```

As the code shows, we can store an integer in the memory given to us by `malloc` using `*`.

We can also write the code more compactly, taking advantage of the fact that we can initialize a variable's value at the same time we declare that variable:

```
int *p = malloc (sizeof (int));
*p = 42; /*Store 42 in the memory we got from malloc*/
```

Finally, compare that code to this code:

```
int q;
int *p = &q;
*p = 42; /*Store 42 in the memory referenced by q*/
```

Here, we are not using `malloc`. Instead, we are letting C manage the memory for one integer, making it available through the variable `q`, and making `p` point to the same memory address as `q`.

In both cases — using `malloc` and using `&` — we are assigning a memory address to a pointer variable.

6 Mallocing Arrays

We know how to ask `malloc` for memory for one integer, and we know how to store an integer in the memory `malloc` gives us:

```
int *p;
p = malloc (sizeof (int));
*p = 42;
printf ("We stored %d.\n", *p);
```

What if we want room to store multiple integers, like 5? We could do that like this:

```
int *q;
q = malloc (5 * sizeof (int));
```

To access the memory for these five integers, we can treat `q` like an array!

```
*q = 2; /*2 is in the first array element*/
*(q + 1) = 48; /*48 in the second element*/
q[2] = 100; /*100 in the third*/
```