COSC 242
Lab Book

Semester two

2020

Department of Computer Science
University of Otago

C Laboratories for Java Programmers

Nathan Rountree and others

Semester 2, 2020

Preface

Welcome to the laboratories for COSC242. We hope you enjoy the material you find here, and that you will help us by pointing out any errors, omissions, or logical inconsistencies you find. With the exception of Appendix B written by Alicia Monteith, Appendix C written by Parviz Najafi, and a few labs written by Iain Hewson, this lab book is the work of Nathan Rountree. It is maintained by Iain, so please send any suggestions or corrections to *ihewson@cs.otago.ac.nz*.

Advice

- *The labs don't all have assessment attached to them; why should I attend?* Learning about data structures and algorithms is challenging—even more so since you will be learning a new programming language. Perhaps the best way to fix the ideas in your mind is to program them. This is like the difference between being *told* how a loaf of bread is made, and actually *making* one yourself—we think it is important to "get your fingers dirty".

  On a more pragmatic note: your internal assessment consists of a programming assignment, one assessed lab, and three practical tests, worth a combined total of 40%. If you attend all of the labs and do all of the programming exercises, you will find these reasonably straightforward. If not...

- *Why do we have to learn C? What's wrong with Java?* We feel that you should know multiple programming languages before you graduate with a degree in Computer Science. Since this course is on data structures and algorithms, we wanted a language that gives the programmer fine control over memory management. We also wanted a language that you will use in the future. We are sure that programming in C is a skill you will never regret learning.

- *I'd like more information on C. Where do I look?* These laboratories assume that you already have some knowledge of Java, and uses that as a springboard into C. When something in C looks the same as in Java (for instance, "for" loops, "while" loops, assignment, the increment operator...) we've not gone into very much detail. This happens a lot, since Java was designed to be easy to learn if you already knew C or C++, and has much in common with them. If you would like to concentrate for a while on the C language (rather than on data structures and algorithms), then *The C Programming Language* by Kernighan and Ritchie is the best book you could hope to find. The first chapter is an excellent tutorial on C.

  On the web, you may wish to check out Steve Summit's home page:

  `http://www.eskimo.com/~scs`

His C FAQ is very useful, as are his course notes.

- *Your labs are awfully wordy.* We have tried to include everything you need to know to complete each lab's exercises. Please read the whole lab *before* you sit down at the computer so that you have a feel for where each one is going. Five minutes of extra reading could well save you several hours of frustrating hacking.

- *I'm finding Unix a challenge. What should I do?* You can make life easier for yourself in several ways. First, make sure that you create a directory for your COSC242 work, and then work in a new directory for each lab. Never throw away any code: you will reuse it in future labs. Next, make friends with your text editor. Learn one new emacs keystroke each day, and use it all day so that you fix it in your muscular memory. Finally, check out the Unix course notes from Ohio State University:

  `http://www.cs.duke.edu/csl/docs/unix_course/intro-1.html`

- *Your code examples don't have much error checking in them.* True. Data structures and algorithms are challenging enough without adding lots of extra code to check for bad data as well. In this course, we assume that every program has a *preprocessor* which does all the checking for malformed files. If you feel it is important to connect the ideas of data structures, algorithms, and good program behaviour with respect to bad data, please consider taking COSC345 (Software Engineering) next year.

Please keep these principles in the forefront of your mind:

1. **Never throw away code.**
2. **Do every lab.**
3. **Test, test, test.**

In these labs you will create pieces of code which will be cut-and-pasted verbatim into later exercises. You will find the later exercises (and assessments) incredibly difficult if you haven't completed the earlier ones, or if you have deleted previously written code. You also need to be sure that the code you have written will work in all of the cases it is likely to meet (assuming good data).

Your demonstrators are allowed to help you with any lab exercise material, but may not write any code for you. If you require help with assignment material, try to put your questions in terms of the lab exercises on which the assignment is based.

We hope you enjoy this course.

The 242 team:
Steven Livingstone
Iain Hewson

Contents

Lab 1

Imperative Programming

1.1 Pseudocode

We're going to start off by treating programs as scripting problems. That simply means describing the solution to a problem as a sequence of events which must occur in their correct order. When a script gets too big, we'll try to collapse it by introducing loops or by removing some of its functions to other scripts. When we want to run a script to see if it really works, we'll translate it to C, compile it, and run the resulting program.

Here's a small script to show you what I mean:

```
main {
  initialise i to 0
  print "Welcome to the script."
  print "Let's print out some numbers:"
  print i
  add 1 to i
  print i
  add 1 to i
  print i
  add 1 to i
  print i
  add 1 to i
  print i
  add 1 to i
  print "Finished."
}
```

We call this a script because it specifies a sequence of steps, just like the script of a movie. Notice that I've given it a name (main) and have used braces to create a block (as we do in Java). Each line contains an instruction to be carried out, and should be reasonably unambiguous.

Sometimes in a script we notice that we've just done the same thing over and over again, so our scripting should allow for repetition:

```
main {
  initialise i to 0
  print "Welcome to the script."
  print "Let's print out some numbers:"
  repeat 5 times {
    print i
    add 1 to i
  }
  print "Finished."
}
```

Sometimes it's convenient for a script to call another piece of script, so we have to give names to pieces of script:

```
print_numbers {
  initialise i to 0
  repeat the next block 5 times {
    print i
    add 1 to i
  }
}

main {
  print "Welcome to the script."
  print "Let's print out some numbers:"
  call print_numbers
  print "Finished."
}
```

This all seems pretty basic after two tough semesters of Java, so why am I making these points at all?

Well, this course is all about data structures and algorithms. A script which calls other scripts and allows repetition is a convenient way to represent an algorithm: especially if you can translate the script into a programming language, compile it, and verify that it runs the way you think it should.

Up until now you have been able to define the "structure" of your data in terms of Java classes. The Java compiler and virtual machine have managed the computer's memory for you so that you can be sure that each object will have the correct amount of room. In C, you have to manage that memory yourself, and clean it up when you're done. Here's an example of scripting which does that sort of thing—it calculates the median of numbers stored in a file:

```
main {
  initialise my_array to be able to hold 16 integers
  while (there are still unread items in datafile) {
    if (my_array is full) {
```

```
      double the amount of memory available to my_array
    }
    put first unread item from datafile into first empty
      space in my_array
  }
  sort the numbers held in my_array
  find the middle number in my_array
  free the memory associated with my_array
}
```

This script uses an old trick to ensure that an array is big enough to hold all the data in a file, even though you don't necessarily know how many items are in the file to start off with.

This technique of using structured English to write scripts is called "pseudocode". You will see examples of pseudocode all through your textbook, since that's how they describe even the most complex algorithms and data structures. Their pseudocode is rather formal though; in these laboratory sessions we'll try to stick with examples such as those above.

Over the labs of this course we're going to look at how to describe memory management ("data structures") and scripts ("algorithms") in C. Sometimes we'll do this by writing a piece of pseudocode and then translating it directly to C; this is a useful way of ensuring that your programs do what you mean them to do. The version of C that we will use is C89. The compiler you will use in these labs (gcc) supports quite a lot of the more recent C99, but not all of it. Occasionally, I'll make a comment regarding C99 in square brackets, [like this].

1.2 Translating Pseudocode to C

C code looks like Java code. If you understand what this means in Java:

```
for (i = 0; i < 10; i++) {
  System.out.println("Line number " + i);
}
```

then it isn't very difficult to work out what this means in C:

```
for (i = 0; i < 10; i++) {
  printf("Line number %d\n", i);
}
```

So it will often be possible to figure out what's going on just by applying what you already know about Java.

Let's take the scripts written above and translate them into C. Here's the second script (which is the same as the first one, but without the silly repeated lines):

```
#include <stdio.h>

int main(void) {
  int i;
```

```
  printf("Welcome to the C program.\n");
  printf("Let's print out some numbers:\n");

  for (i = 0; i < 5; i++) {
    printf("%d\n", i);
  }

  printf("Finished.\n");
  return 0;
}
```

Notice the similarities to Java:

- We have a section called **main**. In both Java and C, the program instructions start executing from the beginning of that routine.
- In both Java and C we use a single line of code (a **statement**) to print something to the standard output. The statement is ended by a semi-colon.

Now the differences:

- The **main method** of a Java program is "void" (has no return value) whereas the **main function** of a C program will return an integer. This is important: C programs return a value to let the operating system know how everything went. On Unix, an exit code of 0 means "everything ok".
- We are used to the **main method** of Java programs taking a String array as an argument. Our C program specifies that there will be no arguments to the program with the **void** keyword. Later on, we will see how to pass arguments to C programs.
- C doesn't care what the file that contains your code is called, so you don't have to worry about your "hello" program being in a file called "hello.c"—but it's still a good idea.

For now, just imagine that the `#include` directive is like an `import` directive in Java. In this case, it's telling us to use the `standard I/O (stdio)` library to print things out.

Exercise:

Put the C code above into a file (whose name ends with `.c`), compile it and run it.

To compile the code, type

```
gcc -W -Wall -O2 -ansi -pedantic -g first-program.c -o print-nums
```

The flags `-ansi -pedantic` ensure that anything other than C89 will be rejected. With the `-O2` flag (that's O for optimise, not a zero) the compiler tries to reduce code size and execution time. The flags `-W -Wall` tell gcc to check for the widest range of possible errors (some checks require optimisation to be turned on). They won't stop the program compiling, but they will warn you that something fishy might be going on. The `-g` flag includes debugging

information in the executable (which could be very useful later). The `-o` flag comes before the name of executable file you want to produce. *Be careful not to put the name of a source file after the -o flag, because it will get overwritten if you do.*

To run the resulting program, type

```
./print-nums
```

Now, at the Unix prompt, type

```
echo $?
```

You should find that you get the number returned by `main()` (currently 0).

Try changing the value returned, recompiling and checking the exit status. What happens with really large numbers? What happens with negative numbers?

1.3 Variables and Loops

Variables in C look remarkably similar to variables in Java. You can declare their existence and assign values to them in just about the same way. However C is slightly picky about one thing: you have to declare any variables you are going to use in a block *right at the top of that block*. [C99 has removed this restriction.]

Loops also look similar to what you are used to in Java, and work pretty much the same way. For loops, while loops, and do-while loops are all available.

Here's some code that should look quite familiar to a Java programmer:

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

int main(void) {

  /* variable declarations */
  int i = 0;
  double j = 1.0;

  /* a familiar "for loop" */
  for (i = 0; i < 10; i++) {
    printf("%d %f %f\n", i, j, sqrt(j)); /* sqrt defined in math.h */
    j *= 2;
  }

  return EXIT_SUCCESS; /* defined in stdlib.h */
}
```

Important: the compilation command for this piece of code is different! You need to link to the math library to get functions declared in math.h, so adjust the compile command to:

```
```

```
gcc -W -Wall -O2 -ansi -pedantic -g using-math.c -o print-sqrts -lm
```

The code for math library functions lives in `/usr/lib/libm.a`, and the `-lm` directive tells the compiler to link that with your program.[1]

We're still scripting here; no objects, no classes, just one thing after the other: start the main function, declare some variables, do the loop, exit the program.

Some points to note:

- You can follow pretty much the same rules for naming variables in C as you do in Java. By convention, we use `my_variable` rather than `myVariable`.
- Assignment to variables and checking for equality has the same trap in C as in Java: `a = b` assigns `b` to `a` whereas `a == b` tests whether `a` equals `b`. Actually it is more of a trap in C because any value can be used as a condition, whereas Java requires that all conditions be boolean values.
- Although it's a convention that a successful program returns 0, we don't have to rely on knowing this. If we include `stdlib.h` in the program, we get the values `EXIT_SUCCESS` and `EXIT_FAILURE` defined for us.
- If we want to do some mathematics, the `math.h` library defines lots of useful functions for us, such as `tan()` and `cos()`.
- "For" loops look just like they do in Java. One small exception: we have to declare the loop variable (`i` in this case) at the top of the function; we can't do it in the loop header. [We can in C99.]
- Anything between `/*` and `*/` is a comment. Comments may extend over several lines. Don't nest comments inside one another because "`*/`" means "anything that follows after this is code". C does not have "`// this sort of comment.`" [But C99 does.]

The `printf` statement deserves a little more attention. `printf` must take at least one argument: a string. Inside that string you may set "place-holders" for expressions—in this case, the "%d" and "%f" parts of the string are placeholders. The list at the end of the string must match the number and types of the placeholders; so if you said

```
printf("%f %d %f", a, b, c);
```

you should be careful that `a` and `c` are of type double, and that `b` is of type integer, since that is what `printf` will be expecting.

Exercise:

1. Put the code example above into a file (remembering to use a **.c** suffix), then compile and run it. Note that the numbers that are printed out don't line up very well.

[1] Not all Unix platforms need an explicit `-lm` to link the math library, but it won't do any harm so it's a good idea to get into the habit of using it.

You can change the placeholders in `printf`'s format string to force the numbers to have a certain field-width or number of decimal places. For instance, `%3d` will set a field-width of 3 and `%8.3f` will set a field-width of 8 with 3 decimal places. You can get a full description of how `printf` works by typing `man 3 printf`.

2. Change the placeholders in the format string to make the output line up nicely.

3. Make a copy of your previous program, calling it something like "fibonacci.c" and alter it so that it prints out the first 40 Fibonacci numbers, all nicely lined up. (That's the series that goes 1, 1, 2, 3, 5, 8, 13, 21, 34...). Here's some pseudocode that may help:

```
main {
  initialise f to 0 and g to 1
  repeat 40 times {
    print g
    store the value of g somewhere else
    add f to g
    place the old value of g in f
    if (we've printed out 5 numbers in a row)
      print a newline character
  }
}
```

1.4 Functions: Calling Other Pieces of Code

So how do we break up the behaviour of a program the way we broke up the script earlier on? In Java, we use **methods** to contain short bursts of code that we want to use over and over again. In C, we refer to a callable piece of code as a **function**.

Let's take the script we looked at earlier (the one that called a subsidiary script to do the counting part) and turn it into a C program:

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

void print_numbers() {
  int i = 0;

  while (i < 5) {
    printf("%d\n", i);
    i++;
  }
}

int main(void) {
  printf("Welcome to the C program.\n");
  printf("Let's print out some numbers:\n");

  print_numbers();
```

```
  printf("Finished.\n");
  return EXIT_SUCCESS;
}
```

Some points regarding functions:

1. Note that once we've defined a function, we can just call it by name. This is similar to the situation in Java in which a class calls one of its own methods.
2. Functions can return values or be void, just as in Java. Return values may be stored in variables, but they don't have to be.

Just so that we can see a wider range of return types and variables used, here's a slightly more complicated example. Note that if you wish to compile this example, you would have to add "`-lm`" to the end of your compile command, since we have called one of the math functions (`sqrt`).

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

int sum_of_squares(int x, int y) {
  printf("Calculating the sum of squares of %d and %d\n", x, y);
  return x * x + y * y;
}

double sum_of_sqrts(double d, double e) {
  printf("Calculating the sum of square roots of %f and %f\n", d, e);
  return sqrt(d) + sqrt(e);
}

void print_bigger_arg(int x, double d) {
  printf("This function doesn't return anything;\n");
  printf("It just prints out whichever argument is bigger.\n");
  printf("The biggest argument is ");
  if (x > d) {
    printf("%d\n", x);
  } else {
    printf("%f\n", d);
  }
}

int main(void) {
  int first_result;
  double second_result;

  first_result = sum_of_squares(3, 4);
  printf("result = %d\n", first_result);
  second_result = sum_of_sqrts(3.0, 4.0);
  printf("result = %f\n", second_result);
  print_bigger_arg(3, 4.0);
```

```
  return EXIT_SUCCESS;
}
```

The `print_bigger_arg` function has an "if" statement in it. It behaves the same as in Java, but the things going on underneath are different. In C there is no Boolean type: "false" and "true" are represented by "0" and "non-0". So, "<" returns 1 for true and 0 for false. [C99 has a Boolean type, available if you include the `stdbool.h` header. It's still implemented as integers, though.]

Exercise

Write a program that prints out the first 100 prime numbers.

Here is some pseudocode that may help you (but you can choose another way of doing it if you prefer):

```
is_prime (candidate) {
  for each number n from 2 up to candidate {
    if (candidate divided by n leaves no remainder) return 0
  }
  return 1
}

main {
  initialise candidate to 2
  initialise num_printed to 0
  while (num_printed is less than 100) {
    if (is_prime(candidate)) {
      print candidate
      increment num_printed
    }
    increment candidate
  }
}
```

Extra for anyone who is keen: design and implement a more efficient way of printing prime numbers.

1.5 Tricks and Traps Summary

† The `main` function returns an integer value to the operating system.

† No nesting comments! (Just like Java)

† All variables declared at the top of each block (including loop variables).

† There is no Boolean type in C89; use 0 for "false" and non-0 for "true".

† Be careful that the arguments to `printf` match their format string, because the C compiler doesn't really care. The `-Wall` flag to the gcc compiler will produce a warning.

† When you use functions from `math.h`, you have to add `-lm` to the compile command to link in the math library.

Lab 2

Manipulating Data

Much of this course is going to involve reading data into your program from a file, doing something clever to it, then printing out the results. Often, we're going to be keenly interested in how quickly this happens depending on the size of the data. Obviously, the next thing we should attack is how to get data into your program and how to process a whole file of the stuff.

2.1 Reading Input

Input is much simpler in C than it is in Java, but that simplicity can lead to danger. Specifically, in C we can write data directly to a particular piece of the computer's memory—including memory that contains the running program's instructions. This would be disastrous, so we must be very careful! Here is an example of reading in an integer and a double using the `scanf` function:

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

int main(void) {
  double my_double;
  int my_int1, my_int2;
  int return_code;

  printf("This program will read in a double and two integers.\n");
  printf("Please enter a floating-point number: ");
  /* make sure output is immediately written to screen */
  fflush(stdout);
  return_code = scanf("%lg", &my_double);
  /* return_code will be 1 if one double was read, 0 if nothing was
     successfully read, or EOF if the end of input was reached. */
  if (return_code != 1) {
     printf("scanf returned code %d\n", return_code);
     return EXIT_FAILURE;
  }
  printf("Now two integers: ");
  fflush(stdout);
  return_code = scanf("%d%d", &my_int1, &my_int2);
```

```
  if (return_code != 2) {
     printf("scanf returned code %d\n", return_code);
     return EXIT_FAILURE;
  }
  printf("Your inputs were: %f, %d, and %d\n", my_double, my_int1, my_int2);

  return EXIT_SUCCESS;
}
```

What's new here?

- The `scanf` function is used to stick input data into variables. Its return value tells you how many items you successfully read according to your format string, and should always be checked. If the return value is `EOF`, then the end of input was reached before anything was successfully read. This scheme can make it hard to recover from errors in bad data, because it doesn't let you try to read the line again. However, `scanf` is perfectly fine if you expect your data to be "clean".
- What's that "&" doing in front of `my_double`, `my_int1` and `my_int2`?

  The "&" operator means "the memory address of". Hence, `&my_double` means "the address of the variable `my_double`." When you declare a variable, C decides which memory cell that variable will live at. The `scanf` function wants to know the address of that cell, not the name of the variable. Imagine that it's saying "I don't care what your name is, I just want to know where you live..."
- It will compile, run, and **crash** if you forget the "&" character in front of the variables. Using `-W -Wall` with `gcc` will warn you of this possibility.

  It compiles because C automatically converts numbers from one format to another when necessary (e.g. when adding an integer to a double). The memory address is just a number, as is the value of `my_int1`. If you forget the "&", `scanf` tries to stick the input into the memory address referred to by `my_int1`, which could be anywhere. Trying to overwrite the wrong piece of memory is to be considered a VERY BAD THING.
- The `scanf` function returns the number of inputs it reads. You can use this to catch errors, or to work out when the data has finished arriving.
- Use `%lg` as the format string for reading in doubles, and `%d` for reading in integers. (That's "percent el gee" for doubles, not "percent one gee".) Note that the `scanf` format codes are not the same as the `printf` codes. If you type `man 3 printf` and `man 3 scanf`, you get a full description of the functions along with their format codes.

Exercise

In a certain figure-skating competition, there are three judges who give out scores from 0.0 to 6.0. The contestant's score is worked out by throwing away the smallest score and averaging the other two. Write a program which accepts three numbers and prints out the competitor's score. It might look something like this:

```
main {
  declare variables s1, s2 and s3 to hold the judge's scores
  read the 3 scores into s1, s2 and s3
  if (s1 is the lowest score) {
    print (s2 + s3) / 2
  } else if (s2 is the lowest score) {
    print (s1 + s3) / 2
  } else {
    print (s1 + s2) / 2
  }
}
```

2.2 Looped input

Most of the time, you are going to write programs that process whole screeds of data and you won't want to type it all in. Let's assume you have the data in a file; here is a way you can always read a single file by redirecting the program's standard input to come from that file:

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

int main(void) {
  double entry = 0.0;
  double total = 0.0;
  int number_of_entries = 0;

  while (1 == scanf("%lg", &entry)) {
    total += entry;
    number_of_entries++;
  }
  /* print the average of all the entries */
  printf("%f\n", total / number_of_entries);

  return EXIT_SUCCESS;
}
```

If you run this program, it will appear to just hang. If you type in a number and press "enter", it just seems to gobble it up. But if you press "ctrl-d" at the beginning of a line, the call to `scanf` returns `EOF` and the loop is exited. ("ctrl-d" is the unix default for "send record now". At the beginning of a line, it sends a zero-length record, signifying end-of-input.)

Rather than typing in each number, we can put all the data in a file and run the program with it:

```
./progname < datafile
```

which tells the program that `stdin` should be redirected to come from the file `datafile`.

What about that while loop? If you read it from the inside out, we are using `scanf` to read a floating point number into `entry`. The `scanf` function returns the number of things it read, so as long as it keeps returning 1 (in this case) we want the loop to keep going. If it returns anything else, the loop will terminate.

Exercise

Back to figure skating again. This time, the program gets a file that looks like this:

```
44  3.1  5.0  4.7
21  4.5  5.1  5.8
...
```

where the first number is each competitor's registration number and the rest are the three judges' scores. Write a program that takes any number of lines of input and tells us who won (*i.e.* who got the largest score). Ignore ties. You can read 4 inputs into 4 variables with

```
scanf("%d%lg%lg%lg", &n, &a, &b, &c)
```

Read the data from the standard input, and redirect the program's standard input to come from the file when you run it. In the directory `$C242/labfiles` you will find a small file called `skating-scores` that you can use for testing purposes.

2.3 Arrays

In Java, an array called `myArray` is a reference to an object that has data fields such as `myArray.length`. In C, the name of an array evaluates to the address of a piece of memory, with *just enough* space to hold the required number of items. If you have 32 bit integers and you request an array of ten integers, C guarantees you only 320 bits of space. This leads to the well-known "buffer overflow" problem, which involves trying to write beyond the end of an array, probably overwriting program instructions.

This piece of code creates 5 random integers in the range 0–4, stores them in an array, and prints them out again with their average:

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

#define ARRAY_SIZE 5

int main(void) {
  int my_array[ARRAY_SIZE];
  int i;
  double total = 0.0;

  for (i = 0; i < ARRAY_SIZE; i++) {
    my_array[i] = rand() % ARRAY_SIZE; /* "%" is "mod", as in Java */
    total += my_array[i];
  }
```

```
  printf("The numbers are: ");
  for (i = 0; i < ARRAY_SIZE; i++) {
    printf("%d ", my_array[i]);
  }
  printf("\n");
  printf("The average is: %f\n", total / ARRAY_SIZE);

  return EXIT_SUCCESS;
}
```

There is one glaringly new thing here: the line that begins with `#define`.

When you're programming in C, you should remember that your program text is processed first by a macro-preprocessor, and only then by the C compiler. When the preprocessor sees a line like `#include <stdio.h>` it finds the appropriate header file and reads it entirely into your program. When it sees a line like `#define ARRAY_SIZE 5` it knows that it should replace all future identifiers `ARRAY_SIZE` with the token `5`. It is a mistake to write `#define ARRAY_SIZE 5;` (with a semicolon) because that will cause `ARRAY_SIZE` to be replaced by "`5;`" all through your program!

Things to bear in mind when using arrays:

1. Arrays are declared like this:

   ```
   int array_of_ints[5];
   float array_of_floats[5];
   char array_of_chars[5];
   ... etc ...
   ```

   In each case we get a slice of the computer's memory just big enough to hold 5 of the items in question. We never need to write

   ```
   int [] arrayOfInts = new int[5];
   ```

   the way we would in Java. The declaration creates the space all by itself, without any call to "new".

2. As in Java, arrays begin at position 0. This means that in an array of size 5, the highest position is 4.

3. Dynamically sized arrays made it into the C99 standard, but are not yet supported by all compilers. For now, we will declare arrays with "literal" values, not variables. Sorry—your array size MUST be known at compile time. There is a way around this that we will look at when we deal with pointers.

4. Two-(and more-)dimensional arrays are possible with declarations like

   ```
   float my_matrix[5][6];
   ```

   which declares a two-dimensional matrix with 5 rows and 6 columns.

5. Most C systems do not check array bounds. In Java, if you have an array of size 5 and you try to stick something in position 10, you get an ArrayIndexOutOfBoundsException. In C, the programmer is considered to be the expert: your program will either go quietly insane, or halt with the message "segmentation fault". In Unix, this means you tried to access memory outside that which your program was allocated.

6. Use a "for" loop to copy an array. C does not allow you to say "`a = b`" if `a` and `b` are arrays.

Exercise

One last fling at figure skating. I suspect that one of my judges is giving a wider *spread* of marks than the others. One way to figure this out is to calculate the average and standard deviation for each judge. The standard deviation is a measure of how far away from the mean each data point tends to be, so for each judge we're going to have to store his/her marks in an array, then work out the average mark. Then we're going to have to go back through the array and see how far away each mark is from the mean, and get a summary of *that*. Unfortunately, the "differences from the mean" average to 0 (since half the values are above the mean and half are below), so we have to *square* each difference, and then take the square root at the end. Here's how to work out the standard deviation for one judge:

```
marks  distance from average  distance squared
-----  ---------------------  ----------------
 3.1    3.1 - 4.24 = -1.14    sq(-1.14) = 1.2996
 4.5    4.5 - 4.24 = 0.26     sq( 0.26) = 0.0676
 5.7    5.7 - 4.24 = 1.46     sq( 1.46) = 2.1316
 5.6    5.6 - 4.24 = 1.36     sq( 1.36) = 1.8496
 2.3    2.3 - 4.24 = -1.94    sq(-1.94) = 3.7636
----                                     ------
21.2 / 5 = 4.24 (mean)                   9.112 / 4 = 2.278 (variance)

Standard deviation = sqrt(variance) = 1.5
```

As it happens, standard deviations are not quite like means since we divide by n - 1 instead of n, where n is the number of data points.

All you have to do is print out something like the following:

```
          Average SD
judge 1 :  4.2   1.5
judge 2 :  5.1   0.2
judge 3 :  5.2   0.6
```

with input exactly the same as the last exercise, *e.g.*

```
44  3.1  5.0  4.7
21  4.5  5.1  5.8
...
```

You can use the same `skating-scores` file from the previous exercise for testing. One last

thing: there will never be more than 10 competitors in the competition, so an `ARRAY_SIZE` of 10 is fine.

To stay with our theme of converting a script to a C program, here is a suggestion for how your pseudocode might look:

```
main {
  declare arrays judge1, judge2 and judge3 to store marks
  while (there is still more input) {
    discard the competitor number
    place each of the scores into its appropriate array
    keep track of how many lines we've read
  }
  calculate the mean for each judge
  calculate the standard deviation for each judge
  print the summary of means and standard deviations
}
```

Remember there is a `sqrt` function in `math.h` for calculating square roots; don't forget to link your program with the math library by compiling with `-lm` at the end of the `gcc` command.

2.4 A Note on Strings

Now that you've met arrays in C, we can talk about strings. In C, a string is nothing more than characters sitting in an array of characters. C's string-handling functions (which you get when you `#include <string.h>`) tell the end of the string by the `\0` character (we call these "null-terminated strings"). Thus, if the array is declared to have n cells, the maximum length of the string is n-1, since one cell is required for the terminator.

2.5 Tricks and Traps Summary

† Don't ever forget the '&' character in front of variables when using `scanf`...

† ...EXCEPT: don't use an '&' character when reading into a string (*i.e.* an array of char—we'll meet strings properly soon).

† Make sure you read into the correct type of variable (*e.g.* don't read a double into an integer). You can check for this with `-Wall` in `gcc`.

† You must set the size of an array at compile time; and you can't use a variable to do it.

† Arrays begin at position 0 and go up to position n - 1.

† You cannot copy an array `x` to another array `y` by writing `y = x`. Use a "for" loop.

† Array bounds are not checked at run-time. Make sure you don't ever overrun an array. This becomes important when reading strings into a fixed-size array.

† A string is just an array of char. If it is declared as length n, it can only store n - 1 characters because of the '\0' character at the end.

Lab 3

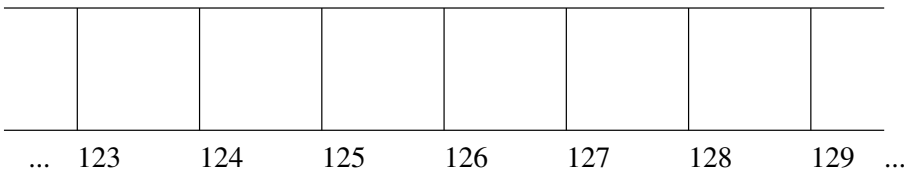
Manipulating Memory

Last time we looked at the problem of getting data from some outside source into your program, and then doing something sensible with it. Today, we're going to look at the same problem but from a different angle: how to organise the *memory* you use to store the data.

Since we're concentrating primarily on the storage itself, you won't see any pseudocode today. However, these concepts are not only relevant to C; controlling the memory your program uses is a general principle that will stand you in good stead no matter what language you use.

3.1 Pointer Variables

Imagine that the computer's memory looks something like this:



with each cell having room for 1 byte (8 bits). This means that a single 32-bit integer will take up 4 of the memory cells; if an integer variable lives at memory location 124, it will run into 125, 126 and 127 as well. The *next* integer would have to live at memory position 128.

We've already met the '`&`' operator, and now it's time to explain what it does. If `i` is an integer variable, then `&i` is the *memory address* of the cell `i` occupies. Simple.

In C there is a particular type of variable designed specially to hold memory addresses. Since a memory address "points" to the start of a variable, these variables are called "pointers". There is nothing scary about them; they are really just numbers that represent memory locations. We can declare a pointer variable like this:

```
int i = 5;     /* a regular old integer variable                */
int *p;        /* a pointer variable called p                   */
int *q, *r;    /* two pointer variables, called q and r         */
int *y, z;     /* a pointer variable y and an integer variable z */
```

You need to be careful when declaring more than one pointer variable on a single line: it's the "*" that makes it a pointer variable. In practice, it's best to stick to one declaration per line.

Here's a short bit of code to illustrate what's going on:

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

int main(void) {
  int i = 5;
  int *p;

  printf("i is %d, the address of i is %p\n", i, &i);

  p = &i;  /* p is now storing the address of i (whatever that is) */
  printf("p is %p, what's stored at p is %d\n", p, *p);

  *p = 6; /* make the value at memory location p 6 */
  printf("what's stored at p is %d, i is also %d\n", *p, i);

  return EXIT_SUCCESS;
}
```

Using "%p" to print the memory address puts it into some sensible format for us (usually hexadecimal); otherwise it's just a mighty big integer. If you want to get rid of the compiler warnings then cast the argument which corresponds to %p to a void pointer[1] like this:

```
printf("i is %d, the address of i is %p\n", i, (void *) &i);
```

The first call to `printf` is self-explanatory—we print out the value of the variable `i`, and then the address in memory where `i` would be found.

The variable `p` is declared as an `int *` (an "int-pointer"), so it can store "the address of `i`". When we print out the value of `p`, we see that it is the same as `&i`.

The third `printf` statement demonstrates a new concept. Sometimes when we know a memory address, we want to know the *value* of what lives there. We can translate "`*p`" as "what lives at memory location `p`". Of course, we already know that "what lives at `p`" is the value 5. This is known as "dereferencing a pointer", because we are finding out what the pointer refers to.

When we write "`*p = 6`", we're saying "whatever is stored at `p`, give it the value 6". Unfortunately, the variable `i` was stored there, so it is now 6 as well.

The trick is to make sure you understand the difference between the *address* and the *value stored there*. If `p` is a pointer-to-an-integer, then `p` is a memory address, whereas `*p` is the actual value stored at that address.

What possible use could this be? Here are the applications of memory addresses which will concern us until the end of this course:

1. In Java you are used to methods which can return values, adjust data fields, but **can't**

[1] To find out more about void pointers take a look at Appendix B at the back of this lab book.

change their arguments. Whatever actual parameters are passed to them are unaffected. The same is true in C—the parameters passed to functions cannot be changed by the function itself. However if we pass *addresses* as parameters, we can change the *values held* by the parameters. Think about how `scanf` works...

2. Pointers let us build up *linked* data structures, which are one of the main topics of this course. A pointer variable at the end of one item in the structure can tell us where to find the start of the next item in the computer's memory.

3. Pointers let us build arrays on the fly, so that we don't have to know how big the array is at run-time. Using functions which allocate memory we can create arrays which are just exactly big enough to store what we want. Using functions which deallocate memory we can "free" arrays we are finished with without having to wait for a garbage collector. This is useful, since we don't *have* a garbage collector by default! (But try Googling for "Boehm garbage collector" for a commonly used garbage collector for C.)

4. When you want a function to operate on a whole array, you don't want that *whole* array to be copied into the function. C knows this, and passes instead the *address* of the first element of the array. Pointers and arrays are so closely related in C that you can still treat the pointer parameter exactly as if it were an array. Here's an example of how an array gets passed to a function:

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

#define ARRAY_SIZE 10

void multiply_by_3(int *a, int n) { /* note: pass int *a rather than int a[] */
  int i;

  for (i = 0; i < n; i++) {
    a[i] *= 3; /* note: can still use [] notation */
  }
}

int main(void) {
  int my_array[ARRAY_SIZE];
  int counter = 0;
  int i;

  while (counter < ARRAY_SIZE && 1 == scanf("%d", &my_array[counter])) {
    counter++;
  }

  multiply_by_3(my_array, counter);
  for (i = 0; i < counter; i++) {
    printf("%d\n", my_array[i]);
  }
  return EXIT_SUCCESS;
}
```

As an extra example of how pointers and memory work in C, here's how to write a function

which swaps the values of its two arguments. It would be impossible to write a function quite like this in Java; you would need to store the two variables as data fields for a method to swap their values successfully.

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

/*
 * "swap" wants the memory addresses of two integers
 */
void swap(int *x, int *y) {
  int temp = *x; /* temp gets the value living at memory address x. */
  *x = *y;       /* the value at x gets the value at y.             */
  *y = temp;     /* the value at y gets the variable "temp"         */
}

int main(void) {
  int a = 3, b = 4;

  printf("a = %d, b = %d\n", a, b);
  swap(&a, &b); /* pass the addresses of a and b */
  printf("a = %d, b = %d\n", a, b);
  return EXIT_SUCCESS;
}
```

Exercise

1. Create a good explanation for why `scanf` needs the "&" character in front of the variables it's going to read into.

2. Write a function which finds the biggest and smallest elements in an array. Test it by calling it from `main`.

   The signature of the function could look like this:

   ```
   void max_min(int *a, int n, int *max, int *min)
   ```

   So the function takes as arguments an array, the length of the array, and the addresses of two integer variables. When the function has finished, the array should be unchanged but the variables whose addresses were passed to `max` and `min` should now hold the largest and smallest values that were in the array.

   The function might be called like this:

   ```
   int my_array[] = { 5, 2, 7, 3, 4 };
   int biggest, smallest;
   max_min(my_array, 5, &biggest, &smallest);
   printf("max value is %d, min value is %d\n", biggest, smallest);
   ```

3.2 Dynamic Arrays

I promised to tell you how to get around knowing the size of arrays at compile time, so here it is. The secret is this: an array name evaluates to the address of the first element of a block of memory big enough to hold the right number of elements. Here's a bit of code to prove it:

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

int main(void) {
  int a[] = { 5, 4, 3, 2, 1 };
  printf("   a: %p\n", a);         /* the array itself */
  printf("&a[0]: %p\n", &a[0]);    /* the address of the first element of a */
  printf(" a[0]: %d\n", a[0]);     /* the value of the first element of a */
  return EXIT_SUCCESS;
}
```

Compile it and run it, and you get output something like this:

```
   a: 0x7fff1d88f580
&a[0]: 0x7fff1d88f580
 a[0]: 5
```

Of course, the addresses that your array name will evaluate to will be different from mine. So if `a` and the address of the first element of `a` are the same thing, surely `int *p = a` would also be the same thing? In fact, if you make the value of `p` the same as the location that `a` evaluates to, you can use `p` *exactly as if it were* `a`, right down to doing things like `printf("%d\n", p[3]);`.

Now imagine you want an array with the size defined by the program's user. You could write something like this:

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

int main(void) {
  int array_size = 0;
  int *my_dynamic_array;
  int i = 0;

  printf("Enter the size of the array: ");
  scanf("%d", &array_size);

  /* initialise the array to the appropriate size */
  my_dynamic_array = malloc(array_size * sizeof my_dynamic_array[0]);
  if (NULL == my_dynamic_array) {
    fprintf(stderr, "memory allocation failed!\n");
    return EXIT_FAILURE;
  }

  for (i = 0; i < array_size; i++) {
```

```
    my_dynamic_array[i] = rand() % array_size;
  }
  printf("What's in the array:\n");
  for (i = 0; i < array_size; i++) {
    printf("%d ", my_dynamic_array[i]);
  }
  printf("\n");

  /* release the memory associated with the array */
  free(my_dynamic_array);

  return EXIT_SUCCESS;
}
```

The main thing here is the call to the `malloc` function (short for *memory allocate*). If the user enters the number 7 and the size of an integer on the machine is 4 bytes long, the call to `malloc` will return a pointer to a piece of memory which is guaranteed to have at least 7 * 4 bytes *reserved—i.e.* that memory will no longer be allocated by any other call to `malloc`, nor will it be used by any local variables until you call `free` with that pointer as an argument. For those who really want to know why the call looks like it does, here's a brief explanation:

The `malloc` function returns a pointer to an allocated number of `bytes`. Now you and I know that on the machines we're using, an integer is 32 bits = 4 bytes wide. However it isn't always the case: on some machines integers are only 16 bits wide, and on others they are 64. Using `sizeof` removes that worry from us, by returning the number of bytes this machine/operating system uses to store an integer.

Note that as soon as we try to allocate memory, we test to see if it worked. This is done by checking that the pointer isn't `NULL`, which is what `malloc` returns if it can't allocate the memory we want. If you were to ask the program for an array of 1000 million integers, `malloc` would return NULL and the program will halt: but ONLY if you make the effort to catch the error!

Our error handling code shows how to halt the program with a return code other than `EXIT_SUCCESS`, but what about the line just before that? Well, Unix programs are expected to send their *proper* output to the *stdout* stream, which is what `printf` does. When something goes wrong, error messages should go to the `stderr` stream, which is what we do here. The `fprintf` function is just like `printf` except that it sends the output to a specified stream rather than `stdout`. There is nothing to stop you specifying `stdout` as the stream used by `fprintf`, but always use `stderr` for error messages.

Exercise

Extend the program above so that it prints out another line—I want to know which numbers have been repeated in my "random" array.

Suggestion: write a function called `display_repeats` which takes the array (and its size) as an argument, and prints out which values in it are repeated (and how many times).

Further suggestion: you know that all your random integers have values between 0 and `array_size - 1` inclusive. You could create an array whose purpose is to keep track of how many times each number appears...

Note: Because pointers can be a difficult subject to understand, we have included an Appendix on this topic at the back of the lab book.

3.3 Tricks and Traps Summary

† `int i = 9; printf("%p", &i);` displays the location of the variable `i`.

† `int *p; p = &i;` stores the location of the variable `i` in the pointer variable `p`.

† `*p = 5;` sets whatever value `p` is pointing at to be 5.

† `int *p, q;` declares ONE pointer variable and one integer. `int *p, *q;` declares TWO pointer variables.

† You must use `#include <stdlib.h>` to get `malloc` and `free`.

† The `malloc` function wants to know how many *bytes* to allocate. Luckily, the expression `sizeof (type)` or `sizeof variable` returns the size of the type or variable in bytes.

† You must `free` any memory you `malloc` *before* you lose the pointer to that memory.

† You must *never never never* `free` memory that you didn't `malloc`.

† You must never `free` the same memory twice.

Lab 4

Sorting Items in an Array

4.1 Assessed Lab: Friday July 17^{th}

Note: This lab is worth 2% of your mark for COSC242. Make sure you bring your ID card to the lab with you.

The cut-off for getting this lab marked is Friday the 17^{th} of July. Since programs often take longer than expected to get working correctly, we encourage you to complete as much as you can before your usual time-slot. There is a note at the end of this lab specifying how it will be marked.

Sorting is the most commonly encountered problem in Computer Science. Over the last 50 years various techniques have been proposed for sorting data, and the analysis of how those sorts behave is an important part of this course. There is one very good way to get a feel for the sorts you will study, and that is to implement them and verify empirically that they behave the way they are supposed to in theory.

We'll start by sorting a file of numbers using a fixed-size array, just to get a feel for the sorting algorithms. In a later lab, we'll look at ways to read in any number of data items—even if we don't know how many items are in the file until we've read them all. That way we can test the sorting algorithms using files of any size without worrying about over-running the program's storage.

4.2 Sorting

This is what we want to do:

```
main {
  while (there are still more items in the file) {
    place the item just read into an array
  }
  sort(the array of items)
  print out the sorted array
}
```

So here's a program that does *nearly* everything we want it to:

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

#define ARRAY_MAX 10

void insertion_sort(int *a, int n) {
    ; /* Sorting code goes here */
}

int main(void) {
  int my_array[ARRAY_MAX];
  int i, count = 0;

  while (count < ARRAY_MAX && 1 == scanf("%d", &my_array[count])) {
    count++;
  }
  insertion_sort(my_array, count);
  for (i = 0; i < count; i++) {
    printf("%d\n", my_array[i]);
  }
  return EXIT_SUCCESS;
}
```

Note that it looks nearly identical to the code in the last lab which multiplied every element in an array by 3. Once again, we don't want to loop from 0 to `ARRAY_MAX`, but from 0 to the number of items in the array (we may have read in *fewer* than `ARRAY_MAX` items).

Exercise

Task: Add the code for insertion sort to the example above. **Note: your compiled program should be called insertion-sort for ease of marking later.**

Details: We may describe insertion sort using English, or we can set down the pseudocode. Here's a bit of both:

Insertion sort works the same way most people sort a hand of cards. We imagine that everything to the left of a certain point is already sorted. We take the first item to the right of that and "pull it out" (leaving a "gap"). We then move everything in the sorted part one place over to the right until a gap opens up at just the right place for us to "insert" what we pulled out. Our left-hand-side is still sorted, but now it is one item longer, and the right-hand-side is one item shorter.

Pseudocode:

```
insertion_sort(int *a, int n) {
   for each position p in array a except the first {
     pull out the item at p and store it in variable 'key'
     move each item that is to the left of position p, and is
        greater than key, one place to the right
     put key in the leftmost vacated position
```

```
   }
}
```

Implement insertion sort by starting with the example code and finishing the `insertion_sort` function. Test your sorting program with 5 numbers to make sure it is sorting properly. Don't forget to test border cases; test it with 0, 1, 10 and 11 numbers as well. Now crank up the `ARRAY_MAX` constant and try it with 100, 1000 and 30000 random numbers.

How do you get a file of random numbers under the BASH shell? Type the following command to get 10 random positive integers:

```
for ((i = 0; i < 10; i++)); do
   echo $RANDOM
done
```

You can get a file of random numbers by redirecting the output of that command with the "$>$" operator.

Exercise

Task: Write a program (separate from your insertion sort program) that implements selection sort. Test it in the same way that you did for insertion sort. **Note: your compiled program should be called selection-sort for ease of marking later.**

Details: Selection sort behaves differently from insertion sort under various conditions. You can use exactly the same code as for insertion sort to get a program running for selection sort (although you should rename the sorting function, of course!).

Here are descriptions of selection sort, in English and pseudocode:

Selection sort assumes that you know how to pick out the *smallest* item in an array of items. Once again, we conceptually break the array into two pieces, left and right. We have n items altogether.

Pick the smallest item in the section from 0 to (n - 1) and swap it with whatever is in position 0. Pick the smallest item in the section 1 to (n - 1) and swap it with whatever is in position 1. Now pick the smallest item in section 2 to (n - 1) and swap it with whatever is in position 2...you get the idea. Stop when you get to position n - 2, since the item now in the last position (n - 1) must be the largest item.

Pseudocode:

```
selection_sort(int *a, int n) {
  for each position p in the array a except the last one {
    find the smallest item from position p to position (n - 1)
    swap the item you find with whatever is at position p right now
  }
}
```

4.3 Timing and Testing your Sorting Algorithms

Timing

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

#define ARRAY_MAX 30000

void insertion_sort(int *a, int n) {
  ; /* code for sorting goes here */
}

int main(void) {
  int my_array[ARRAY_MAX];
  clock_t start, end;
  int i, count = 0;

  while (count < ARRAY_MAX && 1 == scanf("%d", &my_array[count])) {
    count++;
  }

  start = clock();
  insertion_sort(my_array, count);
  end = clock();

  for (i = 0; i < count; i++) {
    printf("%d\n", my_array[i]);
  }
  fprintf(stderr, "%d %f\n", count, (end - start) / (double)CLOCKS_PER_SEC);
  return EXIT_SUCCESS;
}
```

To verify empirically the theory at the centre of this course, we can time how long each sorting algorithm takes. Of course, we expect it to be a function of how many items there are to sort, but we want to ignore such distractions as how long it takes to read in the data, or how long it takes to print it out. Thus, we place timing calls around the call to the sorting function, as shown above.

For this to work, don't forget to `#include` the `time.h` header!

Notice that we send the information regarding timing to the standard error rather than the standard output. This is important: only *results concerned with the purpose of the program* should go to the standard output, since they may be piped along to another program. If this program's job is to sort numbers, we certainly don't want a timing line attached to that output. If we send the two different types of output to the two different streams, we can do this sort of thing:

```
./program > output 2> timing
```

where the standard output ends up in the file `output` and the standard error ends up in the file `timing`. Better yet, if we're really only interested in the timing, we can do this:

```
./program > /dev/null
```

which sends the standard output (*i.e.* the sorted data) to the bit bucket, and leaves the timing information on the screen.

Something to bear in mind is that *timing* a sort is not a very good measure of its efficiency. It depends on how many people are using the machine, what other programs you are running, and of course what type of machine you're using! However, the `clock` call returns *processor* time (the time your program has taken on the CPU) rather than *elapsed* time (the time a user would have to wait to get a result) so is independent of the load on the machine. Furthermore, we're not using the time to say "our sort takes n seconds", but rather "if our sort takes n seconds with p items, we expect it to take m seconds with q items". As such, it's a much less cluttered way of determining scale-up properties empirically than counting swaps or comparisons.

4.3.1 Testing

Now that we have a way of quickly comparing one algorithm with another, we have to think about the sort of tests we want to run on them. Think about what affects the performance of a sorting algorithm—not just the size of the input data, but also the kind of order it may already be in. What constitutes the worst case? The best case? An average case?

Usually we would like to see how a sorting algorithm behaves on 4 types of data: sorted order, reverse order, partially sorted order, and random order. Furthermore, for each of these categories we would like to see how the algorithm behaves with regard to the number of items in the data. So for sorted order we need to produce perhaps 10 files ranging from 3000 items to 30000 items, and the same for reverse order, and so on. This would mean 40 different test runs for each algorithm: not very nice on a GUI, but a piece of cake on a Unix system.

Exercise

First, add code to your selection-sort and insertion-sort programs to display timing information, just as in the example above. Now, in the `$C242/labfiles` directory you will find a shell script called `testsort` for testing and graphing your selection sort and insertion sort programs. Copy it into the same directory as your sorting programs, which should be called "`insertion-sort`" and "`selection-sort`". Run the script by typing `./testsort`.

At the moment, all 8 tests are plotted on one graph, which is really too cluttered. You can change the main parameters at the top of the script to produce a graph for just one sort or for fewer types of data set.

Make sure your two sorting programs are working properly (*i.e.* they successfully sort random integers, including border cases). Now run the testing script to get a graph of your programs' performance, and see if you can answer these questions:

1. What is the chief difference in character between insertion sort and selection sort? If

you didn't know about any other (quicker) sorting methods, under what conditions would you use insertion sort? How about selection sort?

2. How would the tests be affected if we tampered with the maximum number of items to be sorted? What about changing the increment value?

While this is still fresh in your mind, open up the `testsort` script and figure out how it works. In particular, how do we go about generating "partially sorted" data? Can you suggest a better way? You will notice that we don't actually use BASH to generate the random numbers; instead we use 'makenums', a ruby script we used in 241 which is a lot faster and can produce random numbers without duplicates.

To get your marks for this lab you must:

- **Show the demonstrator the files insertion-sort.c and selection-sort.c (make sure they are reasonably well commented)**
- **Run the programs insertion-sort and selection-sort for the demonstrator to show that they really sort numbers correctly**
- **Run the testsort script and show the demonstrator the resulting graph**
- **The demonstrator will then submit your code so that we have an electronic copy of your completed work.**

4.4 Assessment Reminder

You have now completed the first part of the internal assessment for this course worth 2%. The next piece of assessment is:

- the first practical test, which occurs on Tuesday the 28^{th} of July. It is worth 3% of your final mark.

You can find a list of all the internal assessment for this course at the back of this lab book and on the course web page.

Lab 5

More Data Manipulation

5.1 Strings

It's not hard to see how we could extend everything we've done so far to items of type `double`, or any of the other number types (even `char`) that C gives us; after all, we can always make an array of those data types, read them in using `scanf`, and use the < operator to compare and see if they need to be swapped. The task is so trivial, in fact, that we're not going to make you do it.

However, you may recall from Java that *strings* are a different matter entirely. Thinking about Java for a moment, we know that

1. Strings are not a primitive type; there is a Class String that you can instantiate.
2. The < operator won't work with Strings; you have to use the `compareTo` method if you want to know whether one String should come before another in a dictionary.
3. Strings in Java are *immutable*, meaning we can't change the individual character elements "in place". We can call methods which return portions of Strings (*e.g.* the `substring` method) and we can even assign the result of that call to the name of the String we got it from, but there is no method that allows you to "tweak" a couple of characters within a String, or add some characters to a String. For example, there is no `append` method that would let us write:

```
String s = "my dog";
s.append(" has fleas"); // no such method in Java!
```

Strings in C are much simpler beasts. And, like all of C's apparent simplicity, it can lead to danger. Most of the memory allocation errors people make while programming in C are connected with manipulating strings.

A string in C is just an array of `char`. It is almost exactly like an array of `int` with just one difference: the last character in the string is followed by the *null character*, represented by '`\0`' (backslash zero).

5.1.1 Three Types of Character Array

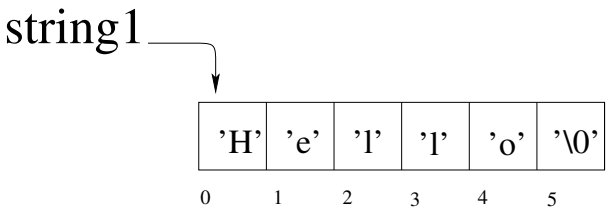
Here's a piece of code to demonstrate three ways of manipulating strings in C:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(void) {
6    char *string1 = "Hello";
7    char string2[10];
8    char *string3 = NULL;
9
10   printf("%s ", string1);
11
12   string2[0] = 'w';
13   string2[1] = 'o';
14   string2[2] = '\0';
15   printf("%s", string2);
16
17   strcpy(string2, "rld");
18   printf("%s!\n", string2);
19
20   string3 = string2;
21   strcpy(string2, string1);
22   printf("%s ", string3);
23
24   string3 = malloc(10 * sizeof string3[0]);
25   strcpy(string3, "world!");
26   printf("%s\n", string3);
27
28   free(string3);
29   return EXIT_SUCCESS;
30 }
```

This program simply prints out "Hello world!" twice, but each of the strings `string1`, `string2`, and `string3` illustrate a different point about character arrays.

First up: on line 3 we include the file `string.h`, which contains useful routines for manipulating character arrays.

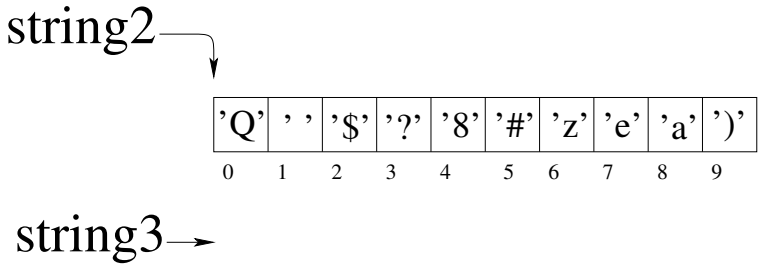
Line 6 shows the declaration of a char pointer that points to a "string literal". The C standard says that *the behaviour of a program that attempts to alter a string literal is undefined*: so you may not write `string1[0] = 'h'`. However, `string1` is like a regular old char array in other respects. The compiler allocates 6 bytes of memory for `string1` to point to (one extra for the `'\0'` that terminates it), and the state of affairs after line 6 is something like this:



Line 7 declares a second string—this time a familiar character array with 10 cells. At this moment, the cells are full of garbage values.

On line 8 we declare a char pointer which currently points nowhere. Since we know how to allocate memory to a pointer and make it behave as if it were an array, this could be convenient if we don't know how big the incoming string is.

The second and third strings can be pictured as looking something like this:

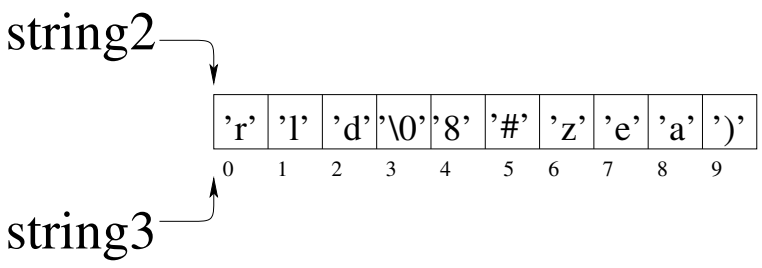


Note that `string2` isn't even null terminated, and `string3` isn't pointing at anything at all. Even *looking* at their contents at this point could lead to strange results.

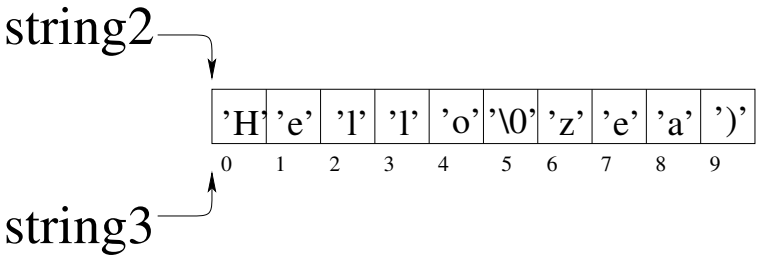
Printing `string1` is no problem: we just use `printf` as on line 10. Before we go about printing `string2` and `string3`, there'd better be something there. Just so you get the idea, on lines 12 and 13 we set the first two letters of `string2`. On line 14, we explicitly terminate that string with a null character—now it's safe to print using `printf`. (Don't ever try to print a string that isn't terminated.)

It's obviously silly to load up strings letter-by-letter, so we call upon the `strcpy` function (defined in `string.h`) to do the work for us in line 17.

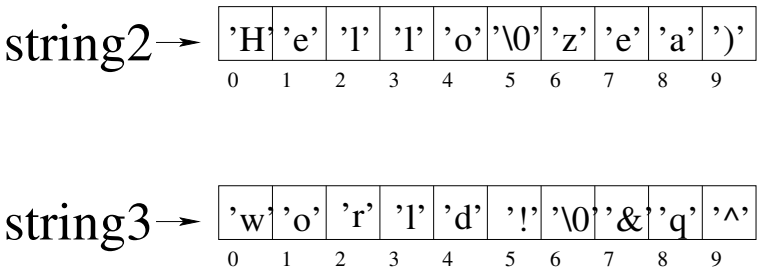
In line 20 we make the char pointer `string3` point to the same place that `string2` points to. This gives us the following situation:



And in line 21 we copy into `string2` those characters in `string1`. Now when we print out `string3`, we get the same characters we would have if we had printed out `string2`. Here's the situation now:



Now we know that we can make a pointer behave just like an array if we allocate memory to it. That's exactly what we're going to do in line 24. After the call to `malloc`, `string3` is pointing at a piece of memory of its very own, so it's now safe to `strcpy` something into it. After line 25, this is what we have:



Note that the characters after the terminator in `string3` are just junk.

5.1.2 Swapping Strings

This is all very well, but which type of string do we want if, for instance, our job is to sort them? Remember, it requires a "for" loop to copy an array (which is how `strcpy` does it) so we can't do the following:

```
char words[100][15]; /* an array of 100 strings, each of size 0 to 14 */
char temp[15];
...
/* do a swap between words[i] and words[j] */
temp = words[i];     /* WON'T WORK */
words[i] = words[j]; /* WON'T WORK */
words[j] = temp;     /* WON'T WORK */
```

However we could do this:

```
char words[100][15]; /* an array of 100 strings, each of size 15 */
char temp[15];
...
/* do a swap between words[i] and words[j] */
strcpy(temp, words[i]);
strcpy(words[i], words[j]);
strcpy(words[j], temp);
```

But that's an awful lot of work just to say "this word should be here", since each `strcpy` copies the word character by character.

The answer is to use char pointers, but be very sure to `malloc` them correctly.

```
char *words[100]; /* 100 char pointers, currently pointing at nothing */
char *temp;
int i;
for (i = 0; i < 100; i++) {
  /* allocate memory to the char pointer at words[i] */
  words[i] = malloc(15 * sizeof words[0][0]);
}
...
/* do a swap between words[i] and words[j] */
temp = words[i];    /* works because just re-positioning pointers */
words[i] = words[j];
words[j] = temp;
```

5.1.3 Other Useful String Routines

If we're going to sort strings, we need to be able to compare them. The function to do this is `strcmp(char *s, char *t)` which expects two null-terminated strings. It returns

- a negative number if `s` comes before `t`;
- zero if `s` and `t` are the same string;
- a positive number if `s` comes after `t`.

If you're going to sort strings, then you'll need to replace the "less than" operator with a call to `strcmp`.

The `strlen` function takes a string as its argument and returns how many characters it found before the terminating `'\0'`. Note that if you pass it a char array of size 80 but with the word `dog\0` in the first 4 positions, `strlen` will return the number 3. Therefore if you are using `strlen` to work out how much memory to allocate for a word, ALWAYS remember to add one for the terminating null.

One frequently asked question is "how do I make up a string out of variables and literals?" The answer is to use the `sprintf` function. This works just like `printf`, but instead of sending the result to `stdout`, it pours it into a string. For example:

```
int i = 113;
char result[80];
sprintf(result, "There are %d items", i);
```

Exercise

1. Here is a program that reads in a pile of words and prints out their average length. Answer the questions which follow it:

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

#define STRING_LEN 80
#define ARRAY_LEN 10000

void *emalloc(size_t s) {
  void *result = malloc(s);
  if (NULL == result) {
    fprintf(stderr, "Memory allocation failed!\n");
    exit(EXIT_FAILURE);
  }
  return result;
}

int main(void) {
  char word[STRING_LEN];
  char *wordlist[ARRAY_LEN];
  int num_words;
  double average;
  int i;

  num_words = 0;
  while (num_words < ARRAY_LEN && 1 == scanf("%79s", word)) {
    wordlist[num_words] = emalloc((strlen(word) + 1) * sizeof wordlist[0][0]);
    strcpy(wordlist[num_words], word);
    num_words++;
  }

  average = 0.0;
  for (i = 0; i < num_words; i++) {
    average += strlen(wordlist[i]);
  }

  average = average / num_words;
  printf("%f\n", average);

  for (i = 0; i < num_words; i++) {
     free(wordlist[i]);
  }

  return EXIT_SUCCESS;
}
```

(a) Notice that we have moved the memory allocation and subsequent error checking code into a separate `emalloc` function. What is the point of doing this?

(b) What is the type of `wordlist[0][0]`?

(c) Why do we allocate `(strlen(word) + 1)` of those rather than just `strlen(word)`?

(d) Is the statement `wordlist[i] = wordlist[j]` correct (assuming `i` and `j` are in-range integers)?

(e) Why do you think there is a `79` between the `%` and `s` in the scanf format string?

(f) Draw a diagram of `wordlist` after the program has read in the words "do", "cats", "eat", "bats".

(g) Check your answers with a demonstrator.

2. Write a program that reads in up to 10000 words and sorts them using either insertion sort or selection sort (your choice). You will need to modify your sorting routine from the last lab to use `strcmp` instead of `<`, and you will need to consider how to control an array of char pointers (HINT: what does an array turn into when passed to a function?). Use an array of char pointers, as in the example above, so that you don't have to use `strcpy` to swap words.

   In the `$C242/labfiles` directory you will find a file of 10000 words randomly selected from `/usr/share/dict/words`. You can use the unix `head` and `tail` commands to pick off 10, 100, 500 words from this list.

5.2 Structs

How do you tie relevant data together in nice chunks the way you can using classes in Java?

For example, imagine you have some industrious person leaning over the side of a boat collecting sea-water and testing it for the amount of oxygen dissolved in it. (Incidentally, such data is of interest to geologists, biologists, conservationists, marine scientists, paleontologists, paleobiologists, and meteorologists.) You may get something like this in Java, if you don't care too much for proper data hiding:

```
class OceanDatum {
  public int x;        // grid-reference east-west
  public int y;        // grid-reference north-south
  public double conc;  // concentration of O2 in mL/L found at grid-ref (x,y)

  public OceanDatum(int x, int y, double conc) {
```

```
    this.x = x; this.y = y; this.conc = conc;
  }
}

public class OceanApp {
  public static void main(String [] args) {
    OceanDatum o = new OceanDatum(2, 3, 4.5);
    System.out.println("o: (" + o.x + ", " + o.y + ", " + o.conc + ") ");
  }
}
```

In C we can create things that *look* like objects, but without methods. Since they impose *structure* on data, they're called `structs`.

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

struct ocean_datum {
  int x;        /* grid-reference east-west                          */
  int y;        /* grid-reference north-south                        */
  double conc;  /* concentration of O_2 in mL/L found at grid-ref (x,y) */
};

int main(void) {
  struct ocean_datum o;

  o.x = 2;
  o.y = 3;
  o.conc = 4.5;
  printf("o: (%d, %d, %.2f)\n", o.x, o.y, o.conc);

  return EXIT_SUCCESS;
}
```

This probably looks a little odd, but kind of familiar. Let's try and see what's going on in terms of what you know from Java.

- The declaration of `struct ocean_datum` is like the declaration of a class in Java. It has 3 data fields, all of which are public (and there is no way to make them private). It has no methods (and can't have any).
- **Unlike** Java, the name of this type is NOT `ocean_datum`; it is `struct ocean_datum`. Always think of structure types with the word `struct` in front.
- Also **unlike** Java, you need a semi-colon at the end of a `struct` declaration.
- Since we've declared the `struct ocean_datum` outside any functions, its declaration is in force from the point of declaration until the end of the file. Later on, we'll see ways of making a declaration known to other files as well.
- All the data fields are public, so we can set them directly using the "dot" notation, without bothering to write modifier methods. Nor do we need accessors to view their values.

Although we don't *have* to have special functions to access the data fields, let's write one anyway in the interests of better program design:

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

struct ocean_datum {
  int x;       /* grid-reference east-west                              */
  int y;       /* grid-reference north-south                            */
  double conc; /* concentration of O_2 in mL/L found at grid-ref (x,y) */
};

void print_ocean_datum(struct ocean_datum *o) {
  /* We need to dereference the struct ocean_datum pointer to print it out,
     which is why we write (*o).x rather than just o.x. Unfortunately,
     *o.x doesn't work, since the "." operator has higher precedence than the
     dereference.
     o->x is shorthand for (*o).x and is a very common idiom.
  */
  printf("%d %d %.4f\n", (*o).x, o->y, o->conc);
}

int read_ocean_datum(struct ocean_datum *o) {
  return 3 == scanf("%d %d %lg", &o->x, &o->y, &o->conc);
}

int main(void) {
  struct ocean_datum my_datum;

  while (read_ocean_datum(&my_datum)) {
    print_ocean_datum(&my_datum);
  }

  return EXIT_SUCCESS;
}
```

Now we have a pattern we recognise: read in the data on an input loop; store each row of data in an "object"; pass the object to a function to do something useful with it (print it out, in this case).

One question: why do we pass the *address* of `my_datum` to the `print_ocean_datum` function? Why not just pass the whole structure? There are several reasons, but the main one is efficiency. When C passes parameters into functions, it makes a copy of them—that's why, although you can change parameters' values inside a function, they don't appear to have changed when the function ends. The function is in fact working on a *copy* of the parameters. Do we really want to copy a whole structure just to print it out? Not really, when we can just tell the function the memory address where it can find the structure.

Exercise

Write a program that reads in a pile of `struct ocean_datums`, sorts them by oxygen content, then prints out the sorted data. You may use selection sort or insertion sort. Here is some code to get you started:

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

#define OCEAN_MAX 41981

struct ocean_datum {
  int x;        /* grid-reference east-west                              */
  int y;        /* grid-reference north-south                            */
  double conc;  /* concentration of O_2 in mL/L found at grid-ref (x,y) */
};

void print_ocean_datum(struct ocean_datum *o) {
  printf("%d %d %.4f\n", o->x, o->y, o->conc);
}

int read_ocean_datum(struct ocean_datum *o) {
  return 3 == scanf("%d %d %lg", &o->x, &o->y, &o->conc);
}

int main(void) {
  struct ocean_datum ocean_data[OCEAN_MAX];
  int num_items;
  int i;

  num_items = 0;
  while (num_items < OCEAN_MAX && read_ocean_datum(&ocean_data[num_items])) {
    num_items++;
  }

  /* sort the data here */

  /* print out the array of structs */
  for (i = 0; i < num_items; i++) {
    print_ocean_datum(&ocean_data[i]);
  }

  return EXIT_SUCCESS;
}
```

You will find the file of ocean data in the `$C242/labfiles` directory. Try testing your program with small pieces of it first.

Two extra things you need to know:

- Suppose you wish to swap the structs contained in positions 0 and 1 of an array. This is fine:

```
struct ocean_datum ocean_data[10];
struct ocean_datum temp;
...
temp = ocean_data[0];
ocean_data[0] = ocean_data[1];
ocean_data[1] = temp;
```

But bear in mind that a lot of copying is being done here; all the bytes for each struct have to be copied. It would clearly be much quicker to store and swap struct *pointers*. You don't have to do that today, but think about how you might do it (you'd need an array of pointer to `struct ocean_datum`, for a start).

- You could write a function that compares two structs in rather the same manner that `strcmp` compares two strings. If you do, I suggest that your comparison function takes as arguments the *addresses* of two structs: otherwise C will make copies of the structs in order to compare them (because of the "pass-by-value" nature of C's parameter passing).

5.3 Tricks and Traps Summary

† Strings are just parts of arrays of `char`, terminated by a `\0` character.

† There is nothing to stop you writing past the end of the character array: BEWARE!

† When you declare a struct, you must end the declaration with a semi-colon.

† The data fields of a struct are always public.

† You can't create methods that belong to structs. Instead, you should define functions which take struct addresses as arguments, then manipulate the data fields as necessary.

Lab 6

Recursive Functions

In this lab we're going to look at two things: how to implement recursive calls and how to process sections of arrays separately. On the way we'll look at passing command-line arguments to your programs and processing files in addition to the standard input.

6.1 Simple Recursion

Recursive function calls need not be at all scary. Consider this code:

```
/* n is the size of the array */
void print_array(int *a, int n) {
  if (n > 0) {
    printf("%d\n", a[0]);
    print_array(a + 1, n - 1);
  }
}
```

This is an alternative to the "for" loop that we usually use to print an array. We don't have to use this idea to print a real array; we could use it to transfer a shopping list on paper to a whiteboard. It might go something like this:

1. If there is nothing in the list, do nothing and stop.
2. Transfer the thing at the top of the list to the whiteboard and cross it off the list.
3. Go to step 1.

Exercise

Make a copy of one of your sorting programs. Near the end of the program, you probably have a "for" loop that prints out the newly sorted array. Replace the loop with a call to a `print_array` function like the one above.

6.2 Designing Your Recursion

Printing out an array using a recursive call demonstrates a fundamental principle of programming with recursion: take care of the stopping case first! In this case, we don't want to do anything at all if the array is all "used up" (i.e. `n` has reached zero) so we could have written it like this:

```
/* n is the size of the array */
void print_array(int *a, int n) {
  if (n == 0) {
    /* do nothing */
  } else {
    printf("%d\n", a[0]);
    print_array(a + 1, n - 1);
  }
}
```

But the first version is tidier.

Let's look at another example. The "factorial" of a number n is defined as the product of all the natural numbers from 1 to n inclusive. Another way of looking at that is to call our factorial function "f" and observe that $f(0) = 1$, and $f(n) = n \times f(n-1)$. We have a "stopping case" (which we should program first) and a "keep going" case (which should be in a different branch of code to the stopping case).

You should convince yourself that this is true. Try tracing the recursion of $f(5)$ on paper to see that when you hit the stopping case, the "$\times$" operator successfully stitches up the result.

Exercise

Complete the following program:

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

int factorial(int num) {
  ...
}

int main() {
  int num;

  while (1 == scanf("%d", &num)) {
    printf("%d\n", factorial(num));
  }

  return EXIT_SUCCESS;
}
```

At what level of input does it "blow out"? (Don't worry, it doesn't take long—you can try it by hand!)

6.3 Binary Search

Why this sudden interest in recursive calls? We're trying to get you ready for an implementation of mergesort, which uses recursion to split up, sort, and merge subarrays. As practice, we're going to do a simplified binary search. Remember that binary search is an algorithm which you could use to find a word in a dictionary (a sorted list of words) like this:

1. If our list has no words in it we haven't found it and are finished.
2. Let m be the middle word in our list.
3. If m is the word we are looking for then we have found it and are finished.
4. If m is greater (alphabetically) than the word we are looking for then go to step 1 using the words less than m as our new list.
5. Otherwise go to step 1 using the words greater than m as our new list.

Write some pseudocode to perform binary search on an array of integers using the points below as a guide.

Normally, a binary search would return the position of the element you are searching for. You just need to return true if the element is in the array, and false otherwise. We don't have a Boolean type in C89, so we use the value "1" for "true" and "0" for false.

So, it's obvious what we do if the length of the array is zero: we have no more array to search, so bail immediately with a "fail" result. Also if the "middle" element we look at is the one we actually want, then great: return a "successful" result. That's our two stopping conditions dealt with.

How about our "keep going" conditions? We also have two of those: one for when the item we're looking for is less than the one at the middle position and one for when the item is greater. In the first case, we want to return the result of searching the first half of the array, and in the second, we want the result of searching the second half. In neither case do we want to include the middle element in the subsequent search.

We know in C for an array 'a' that `a` and `&a[0]` may both be treated as the address of the first element of the array. We also know that `&a[n]` is the address of the $(n+1)^{th}$ element, as is `a + n`.

So each time we recursively call binsearch we give it a base address (array + offset, or array[offset]), a length, and the target we are searching for.

It's a good idea to draw a few pictures when deciding which indexes to include and which ones to exclude when call binsearch again with successively smaller portions of the array.

Once we get to an implementation of mergesort, we'll use the same technique of specifying a base address and a length to say which part of the array we want to sort and merge.

6.4 C Stuff: Program Arguments and Files

6.4.1 Program Arguments

There is some new C stuff in the following exercise that you will need to get to grips with before anything else. The first is using program arguments (specified on the command-line), which are demonstrated in the following piece of code:

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

int main(int argc, char **argv) {
  int i;

  printf("There %s %d program argument%s:\n",
         argc == 1 ? "is" : "are",
         argc,
         argc == 1 ? "" : "s");

  for (i = 0; i < argc; i++) {
    printf(" %s\n", argv[i]);
  }

  return EXIT_SUCCESS;
}
```

It turns out that the `main` function doesn't have to have a void parameter list after all; it can in fact take two arguments supplied by the operating system: an integer saying how many arguments there are (`argc` for "argument count") and an array of strings in which the arguments are stored (`argv` for "argument vector"). Convince yourself that `char **argv` is equivalent to `char *argv[]`. You will see both versions in C programs, but I prefer the first because it is honest about arrays being converted to pointers when passed as parameters.

There is always at least one program argument: the name with which the program was invoked; so `argc` is always at least 1, and `argv[0]` is always going to be the name of the program. Accessing `argv[n]` when there are less than $n + 1$ arguments is an error (just like any other out-of-bounds access to an array).

There is one other thing sneakily popping up here: the "?" operator, which behaves the same way it does in Java. The example makes it obvious what it is doing: if you didn't have it, you would have to go through a cumbersome if-then-else branch to get the same grammatically precise effect. You may use it in your programs if you wish, but preferably only when you have a clear two-way choice. In general, the "?" operator works like this:

```
/* suppose we have variables x, y and z. */
/* rather than writing: */

if (y > 10) {
  x = y;
} else {
  x = z;
}
```

```
/* or even */

if (y > 10) x = y;
else x = z;

/* we can write */

x = y > 10 ? y : z;
```

So what's happening is the condition on the left of the `?` is evaluated. If it evaluates to true (non-zero) then the thing on the left of the colon is returned; otherwise the thing on the right is returned.

6.4.2 Files

The following program shows how to open and read a file consisting of integers:

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

#define ARRAY_SIZE 10000


int main(int argc, char **argv) {
  FILE *infile;
  int my_array[ARRAY_SIZE];
  int num_items;
  int i;

  if (NULL == (infile = fopen(argv[1], "r"))) {
    fprintf(stderr, "%s: can't find file %s\n", argv[0], argv[1]);
    return EXIT_FAILURE;
  }

  num_items = 0;
  while (num_items < ARRAY_SIZE &&
         1 == fscanf(infile, "%d", &my_array[num_items])) {
    num_items++;
  }

  fclose(infile);

  for (i = 0; i < num_items; i++) {
    printf("%d\n", my_array[i]);
  }

  return EXIT_SUCCESS;
}
```

Here we are providing a file name as the program argument in `argv[1]`. The object "`infile`" is of type `FILE *` which, under Unix, is pronounced "stream".

To initialise the stream, we call the function "`fopen`" which requires two arguments: a name for the file (can be a full path or a path relative to the directory in which the program is being run). The second argument is a string which indicates how you wish to use the stream ("`r`" for reading, "w" for writing, etc.).

Note that `scanf("%d", &x)` is precisely equivalent to `fscanf(stdin, "%d", &x)`.

Note also that once you have finished reading a stream there is nothing to stop you from then reading the standard input. This can be a convenient way of testing out data structures: fill up the data structure from a file, then test it by reading data on the standard input.

Finally, it is good practice to `fclose` any file that has been opened.

Exercise

Take a copy of one of your sorting programs and make the following changes to it:

1. Allow a filename to be specified on the command-line. Open this file, read it into the array, and sort it. Do not print it back out.
2. Write an input loop for the standard input which reads in an integer, searches for it in the sorted array, and prints a "+" if it is there and a "-" if it is not.

Watch out—don't name your search function "`bsearch`", because there is already a `bsearch` function in `stdlib.h`.

For instance, suppose I name my program `binsearch` and I have two files: `data` and `query`. In the data file I have:

```
3
1
6
4
9
4
7
1
8
```

and in my query file I have:

```
4
5
6
7
10
-3
8
```

This is what should happen (the $ symbols are just supposed to represent the Unix prompt):

```
$ ./binsearch data < query
+
-
+
+
-
-
+
$
```

6.5 Tricks and Traps Summary

† Take care of the stopping case first!

† If your program appears to have got into an infinite loop, it's probably due to the stopping case never being reached. Be sure that the "keep going" cases have some alteration to them that means they are sure to eventually reach a stopping case (*e.g.* make sure the size of the array in a "keep going" call is able to get smaller).

† The "?" operator is tricky; only use it when you are sure you understand precisely what it is doing. Test whether you understand by deciphering the following:

```
printf("%s\n", x < av - stdev ?
               "low" : x > av + stdev ? "high" : "medium");
```

† Don't be fooled into trying to read or open files in some fashion other than what you saw today. The general pattern is:

```
if (opening the file fails)
  Handle the failure
while (reading into variables from the file succeeds)
  process the variables
```

† Routines such as binary search are terribly sensitive to off-by-one errors and problems due to repeated values. Test extensively!

† You shouldn't use the name `bsearch` for your binary search function, because it is already used in `stdlib.h`.

Lab 7

Practical Test I: Finding Longer-Than-Average Words

7.1 Practical Test: Tuesday July 28^{th}

Note: This lab is assessed and is worth 3% of your total mark for COSC242. Each lab stream has been divided up into 35-minute slots, during one of which you will sit the practical test.

Please make sure you come to the lab, with your ID card, and make sure you are on time.

Purpose

Almost all of the labs that we get you to do during this course are formative in nature; i.e. you learn new things and acquire new skills during the course of completing them. This lab is a bit different. Instead of being formative it is summative in nature. It allows you (as well as us) to look back and see what knowledge and skills you have acquired so far.

7.2 Writing a program from scratch

In the previous assessed lab you could complete the work at any time before the deadline, using whatever resources you wanted to. In this lab you must complete the task during a 30-minute time slot, with minimal resources at your disposal. Using only a terminal window, a blank text-editor, gcc and the system man/info pages, you must write a program from scratch that meets the specifications given below.

At first you might think this sounds a bit daunting; however we are *not* asking you to create your program for the first time while sitting the test. We encourage you to write your program to begin with using whatever resources you need. As an initial outline you could use the program on page 42 of lab 5. Once you have done this, and are sure that your program is working correctly,[1] try to write your program again without using any resources (you might need to peek occasionally). Keep doing this until you can write your program com-

[1] Please come and discuss your solution with Iain if you are unsure about anything.

pletely unaided. If you prepare well for this lab then you will find it pretty straightforward. Writing a program like this, without any outside assistance, will build your confidence and competence to tackle more demanding programming tasks.

Note: You are not permitted to access your home directory, nor any other files or computers, nor may you use the internet during this lab.

7.3 Marking

You must complete your program and get it marked within your 30-minute time slot. In the unlikely event that you don't successfully complete your program within the given time you will have the opportunity to do the test again during another slot or during a different lab stream. If you are not confident of your ability to pass it is recommended that you come to the early stream so that, if necessary, you will have a gap between your first and second attempt. Your program must compile without warnings using the `-W -Wall -O2 -ansi` and `-pedantic` flags to gcc.

What you need to do

Your program needs to perform the following tasks.

1. Read, from *stdin*, up to 100 whitespace separated words, and store them in an array of char pointers.
2. Use an emalloc function to allocate memory for each word as it is read in.
3. Calculate the average word length.
4. Print, to *stdout*, each word which is greater than the average length, in the same order they were read in, using a *recursive* function (like the example at the start of lab 6).
5. Print, to *stderr*, the average word length to *two* decimal places.
6. Deallocate all of the memory you allocated.

If no words are entered then there should be no output at all. You can assume that no word will contain more than 79 characters (don't do this when writing real programs!!). To save time, you do not need to comment your program.

There is a script called `checkprac1` which you can run to check that your program is working correctly for a number of different input files. You can examine the test files if you wish to (their location is displayed when the script is run). Note that this only checks your program's output. It doesn't check that you deallocate memory, print recursively, use emalloc etc.

Once your program is finished and passes the test script, raise your hand and a demonstrator will check your work and mark you off.

Lab 8

Mergesort

After doing the previous labs, you have all the tools in place to implement a *real* sorting algorithm: real in the sense that it is really used in commercial applications. We shouldn't forget about insertion or selection sort though, since they are *faster* than mergesort given a sufficiently small amount of data. In the next lab, we'll change our mergesort so that it "drops" to insertion sort for small arrays. For now, we'll just concentrate on getting mergesort working.

Exercise

Implement mergesort for arrays of up to 100000 integers.

Hints

- Use the same structure of program that you used for selection sort and insertion sort (i.e. the one exemplified by the timed sorting program on page 34).
- Your textbook describes mergesort using three arguments: an array and the two indexes between which you want to sort. This is not very C-like, so I suggest you do it differently: use an array and a *length* to specify what you want sorted.
- Your sorting routines have so far taken only two arguments: an array (or, more precisely, an int pointer) to be sorted and an integer specifying how many items to sort. For mergesort, you need an extra argument: an extra integer array to act as a *workspace* for merging. Your mergesort function should therefore look something like this:

```
void merge_sort(int *a, int *w, int n) {
  ...
}
```

  where `a` is the array to be sorted and `w` is the workspace to be used for merging.
- Assuming that you will pass the merge workspace as an argument, here is some pseudocode for mergesort:

```
merge_sort(int *a, int *w, int n) {
  /* take care of stopping condition first */
  if the array to be sorted has fewer than two elements then return

  call merge_sort on the first half of array a
  call merge_sort on the second half of array a

  merge the two halves of array a into array w
  copy array w back into array a
}
```

This looks nice and easy and it is, as long as you are careful and plan every detail!

- Remember, an integer pointer and a length are enough to specify an array to be sorted. So the "first half of the array" is easy to specify: `a, n / 2`. (Note that this is simpler than in binary search, since we aren't trying to exclude the middle element.) The second half of the array is slightly trickier; we want a pointer to the middle integer (`a + (n / 2)`) and the correct length (`n - (n / 2)`).

- Copying the result in `w` back to `a` requires a "for" loop, since C doesn't allow `a = w` if `a` and `w` are arrays. Using a temp pointer to "swap" them won't work either, since the pointers have been "called by value", and will revert to their original values after the function has returned. That means you can make them point elsewhere *within* the function they're passed to, but at the end of the function they'll be back pointing at what they used to be pointing at.

- The hardest part to get right is the "merge". I suggest setting two integers (call them `i` and `j`) which are the array positions of the things to be merged. Thus `i` should be responsible for the section of the array up to (but not including) position `n / 2`, and `j` should be responsible for positions `n / 2` up to (but not including) `n`. You'll need a *third* variable to keep track of which position in `w` the smaller of `a[i]` and `a[j]` should be placed.

  Don't forget that after you've run out of items to merge in one half of the array, there could still be items left in the other half! Make sure that `i` and `j` have *reached* the end of their sections, and if they haven't, copy over the rest of the items to `w`.

- Perhaps some pseudocode will make the merge a little clearer.

```
merge(int *array, int *workspace, int len) {
  initialise indices to point to the beginning of
     the left and right halves of array
  while there are elements in both halves of array {
     compare the elements at the current left and right indices
     put the smallest into workspace and increment both the index
        it was taken from, and the index in workspace
  }
  add any remaining elements from left half of array to workspace
  add any remaining elements from right half of array to workspace
}
```

A Note on Work Time

It is hard to say how long writing this piece of code will take you. Some people may take longer than the time available in the lab, in which case: don't let it bother you—the next lab is an extension of this one and should be quite short if you have a working version of insertion sort. You can use some time in that lab to get this one finished off.

On the other hand, for some people mergesort will just fall right into place and they may be wondering what to do next. Here is a suggestion:

Use the script from Lab 4 to test mergesort against, say, insertion sort. At what number of items does the difference become obvious? How many items do you need to test mergesort on before its upward curve becomes apparent? Is the line smoother or bumpier than the lines for insertion sort? Why?

Lab 9

Flexible Arrays

Up until now, we have been specifying our array sizes at compile-time. This is obviously not satisfactory, given than we shouldn't have to recompile our program just because we've decided to add a few more items to our data file. To solve this problem, we'll begin using the `realloc` function, which can increase or decrease the amount of memory allocated to a pointer variable. This introduces some new hassles such as keeping track of how much space you've got and how much you've used up; I'll suggest some strategies for dealing with that sort of thing.

9.1 Reallocation

The following piece of code demonstrates the use of the `realloc` function. It uses a trick mentioned in the very first lab to ensure that your array is never overrun: simply double the array's capacity whenever the array is full. Of course, we only double the capacity if the array is full AND we'd like to insert another item.

The `realloc` function differs slightly from `malloc`. Instead of only taking a size as an argument, it takes the pointer that you wish to reallocate and a size. It returns a (possibly different) pointer whose contents up to the minimum of the old and new sizes are unchanged (in other words, it copies across contents if necessary).

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

static void array_print(int *a, int n) {
  int i;

  for (i = 0; i < n; i++) {
    printf("%d\n", a[i]);
  }
}

int main(void) {
  int capacity = 2;
  int itemcount = 0;
  int item;
  int *my_array = malloc(capacity * sizeof my_array[0]);
```

```
  if (NULL == my_array) {
    fprintf(stderr, "memory allocation failed.\n");
    exit(EXIT_FAILURE);
  }

  while (1 == scanf("%d", &item)) {
    if (itemcount == capacity) {
      capacity += capacity;
      my_array = realloc(my_array, capacity * sizeof my_array[0]);
      if (NULL == my_array) {
        fprintf(stderr, "memory reallocation failed.\n");
        exit(EXIT_FAILURE);
      }
    }
    my_array[itemcount++] = item;
  }

  array_print(my_array, itemcount);
  free(my_array);

  return EXIT_SUCCESS;
}
```

Exercise

Add an `array_sort` function to the code above and call it before the `array_print` function. Now you have a program that can sort any number of integers, bounded only by the amount of virtual memory the operating system is willing to give to your program. You may use any sorting technique you like (but insertion sort is sufficient).

9.2 Code Modules

When we looked at the `malloc` function back in lab 5 we moved the memory allocation and error checking code into a separate function called `emalloc`. Here is a useful principle that may help you write better code: "where possible, keep housekeeping out-of-sight." In the code above, we checked the return values from `malloc` and `realloc` to make sure we hadn't run out of memory. This distracts us from the main purpose of the program, and makes for some unnecessary reading for someone else who may be trying to understand the program. It would be nicer if we could do something like this:

```
int capacity = 2;
int *my_array = emalloc(capacity * sizeof my_array[0]);
... /* go on with the program */
```

where `emalloc` is an error-checking version of `malloc`. If we had such a function in a place that was easily available to us, we'd use it all the time. A similar thing would be nice for `realloc`.

Here's what the new program would look like:

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

int main(void) {
  int capacity = 2;
  int itemcount = 0;
  int item;
  int *my_array = emalloc(capacity * sizeof my_array[0]);

  while (1 == scanf("%d", &item)) {
    if (itemcount == capacity) {
      capacity += capacity;
      my_array = erealloc(my_array, capacity * sizeof my_array[0]);
    }
    my_array[itemcount++] = item;
  }

  /* do something useful (e.g. sorting) here */

  free(my_array);

  return EXIT_SUCCESS;
}
```

This is much nicer! We are beginning to separate our code into different *modules* (or files)[1] so that we can easily reuse code, and not have duplicated code in our programs. What we are actually doing is reading in a header file (`mylib.h`) that contains function prototypes for `emalloc` and `erealloc`. Since we place double quotes around it rather than angular brackets, the preprocessor knows to look in the current directory rather than in `/usr/include`. The preprocessor replaces the line

```
#include "mylib.h"
```

with the actual contents of `mylib.h`. The `mylib.h` file looks like this:

```
#ifndef MYLIB_H_
#define MYLIB_H_

#include <stddef.h>

extern void *emalloc(size_t);
extern void *erealloc(void *, size_t);

#endif
```

What is new here?

[1] The C standard refers to what we are calling modules as translation units.

- As you can see, a function prototype is like a function with no body. It simply specifies what the function will return and what the types of its arguments should be.
- The preprocessor directives:

```
#ifndef BLAH_BLAH_
#define BLAH_BLAH_
...
#endif
```

  ensure that no matter how many modules read in the definitions in `mylib.h`, they will only be defined *once* per module. The token that gets defined doesn't actually matter, as long as it is unique to the library. Traditionally, C programmers use `#ifndef X_Y_` for any file `x.y`.
- The `extern` keyword says that the definition is going to hold beyond the boundaries of the file in which it is written. The opposite idea holds when you declare a function `static`, which restricts its use to just the file in which it is written.
- We are including a new header file (`stddef.h`) since both functions take a parameter of type `size_t` (an unsigned integer returned by the `sizeof` operator).

But how does the program know what to DO when `emalloc` or `erealloc` is called? Well, we have a *specification* file (that's `mylib.h`) so what we need now is an *implementation* file (`mylib.c`).

```
#include <stdio.h> /* for fprintf */
#include <stdlib.h> /* for size_t, malloc, realloc, exit */
#include "mylib.h"

void *emalloc(size_t s) {
  /* implementation of emalloc goes here */
}

void *erealloc(void *p, size_t s) {
  /* implementation of erealloc goes here */
}
```

Note that in `mylib.c` we include `mylib.h` so that our implemented functions will be forced to match our specified functions.

Exercise

Implement `emalloc` and `erealloc`. Alter your "unlimited items" sorting program so that it reads in `mylib.h` (use `"mylib.h"` rather than `<mylib.h>` so that the current directory is searched rather than the system headers). Your implementations should be in `mylib.c`.

You can compile the new program like this:

```
gcc -ansi -pedantic -W -Wall -O2 -g -o program program.c mylib.c
```

But if you want to be more explicit about it, try these commands:

```
gcc -ansi -pedantic -W -Wall -O2 -g -c mylib.c
gcc -ansi -pedantic -W -Wall -O2 -g -c program.c
gcc -o program program.o mylib.o
```

This method is more honest about what is really going on: `mylib.o` can be generated separately, as can `program.o`, without actually being able to see `mylib.o` (it gets all the information it needs from `mylib.h`). The last stage *links* the two object files to produce the final program.

9.3 A Flexible Array ADT

Take another look at the `main` function now that `emalloc` and `erealloc` have been moved out to `mylib`. It still isn't ideal; see how the internal details of the data structure are exposed. We have a variable to represent capacity, another to represent itemcount, and it's possible to read or alter them at any time. Not only is this not very safe, it's very inconvenient if we want *two* arrays in the program: we would have to have something like

```
int array1_capacity = 2;
int array1_itemcount = 0;
int array2_capacity = 2;
int array2_itemcount = 0;
```

which quickly starts to get ugly. What we'd really like is to be able to write something like this:

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

int main(void) {
  int item;
  flexarray my_flexarray = flexarray_new();

  while (1 == scanf("%d", &item)) {
    flexarray_append(my_flexarray, item);
  }

  flexarray_sort(my_flexarray);
  flexarray_print(my_flexarray);
  flexarray_free(my_flexarray);

  return EXIT_SUCCESS;
}
```

The "flexible array" is looking after its own capacity, itemcount and storage. Since these types are int, int, and int *, we must be talking about our flexarray being some sort of `struct`. Since the flexarray is being passed as a parameter to flexarray-manipulating functions, and

since we don't like passing structs around if we can help it, our flexarray type is in fact a *pointer* to a struct. Here is the `flexarray.h` file:

```
#ifndef FLEXARRAY_H_
#define FLEXARRAY_H_

typedef struct flexarrayrec *flexarray;

extern void      flexarray_append(flexarray f, int item);
extern void      flexarray_free(flexarray f);
extern flexarray flexarray_new();
extern void      flexarray_print(flexarray f);
extern void      flexarray_sort(flexarray f);

#endif
```

Note that we don't have to specify the `struct flexarrayrec` in `flexarray.h`. Instead, we can specify it in `flexarray.c`, which hides the internals from the main program. In this way, we get a little bit of the data hiding safety that we're used to in Java.

What assumptions do we appear to be making in our new improved program, and what implications do they have for the design and implementation of our flexarray module?

1. It seems as though we should not use a flexarray until it has been set up correctly by a call to `flexarray_new`. If we wanted two flexarrays, we'd have to do something like this:

   ```
   flexarray f1 = flexarray_new();
   flexarray f2 = flexarray_new();
   ...
   ```

   after which we would assume that `f1` and `f2` are separate objects, possibly with differing itemcounts and data.

2. There doesn't appear to be any programmer-imposed limit on the number of items we can fit in the flexarray. I guess flexarrays must be taking care of their own details regarding capacity and itemcount, and `flexarray_append` is reallocating memory as necessary.

3. We don't know (and shouldn't have to care about) what sorting algorithm the `flexarray` module is employing. The best sorting technique we have encountered so far is mergesort, so perhaps we should use that. However, we may discover an even better one in the future—and it wouldn't make any difference to the main program at all. All we would have to do is modify the flexarray package and re-link the program (assuming the specification of `flexarray.h` remained the same). Software engineers consider this to be a **good thing**.

This beast we have created is an Abstract Data Type, or ADT. An ADT is implemented by a data structure that only allows us to communicate with it via its public interface (the functions in `flexarray.h`). That interface defines fully the behaviour of the ADT. In this case, our ADT behaves like a flexible array, so we call it a "flexarray ADT". Often, ADTs

are allowed to store any type of data (not just integers, as here); however that is a separate issue which we call *genericity*. This is a topic beyond the scope of this course, but it is not impossible to implement in C: we can either use "void pointers" (`void *`) to point to the objects we wish to store, or we can use the preprocessor to give us "template code" which allows us to declare a `container_of_int` or a `container_of_double`.

Exercise

Create the implementation file, `flexarray.c`. Here's a start:

```
#include <stdio.h>
#include <stdlib.h>
#include "mylib.h"
#include "flexarray.h"

struct flexarrayrec {
  int capacity;
  int itemcount;
  int *items;
};

flexarray flexarray_new() {
  flexarray result = emalloc(sizeof *result);
  result->capacity = 2;
  result->itemcount = 0;
  result->items = emalloc(result->capacity * sizeof result->items[0]);
  return result;
}

void flexarray_append(flexarray f, int num) {
  if (f->itemcount == f->capacity) {
    /* do the old "double the capacity" trick */
  }
  /* insert the item in the last free space */
}

void flexarray_print(flexarray f) {
  /* a "for" loop to print out each cell of f->items */
}

void flexarray_sort(flexarray f) {
  /* mergesort would be good */
}

void flexarray_free(flexarray f) {
  /* free the memory associated with the flexarray */
}
```

Use your flexarray package in a program similar to the example on page 64. It should read in any number of integers from standard input, sort them, and write the sorted flexarray to the standard output.

Hints:

- Recall that the expression `x->y` suggests that `x` is a pointer to some sort of `struct` that has a data field called `y`. Every time you wish to manipulate the itemcount or capacity of the flexarray in a flexarray-manipulating function, it will pay to remember that!
- Suggestion: use mergesort to sort the flexarray. Don't forget that mergesort requires a workspace for merging. I suggest that you create a workspace just big enough (since you have access to `f->itemcount`, pass it to `mergesort`, then `free` it before the `flexarray_sort` function ends. Your mergesort function should be declared `static`, since it shouldn't be available from outside the `flexarray.c` file.

Lab 10

Quicksort

Exercise

Take your `flexarray.c` file from the last lab and change the sorting routine so that it uses quicksort rather than mergesort. Quicksort has the advantage of sorting "in place", not requiring a buffer of size n to sort n elements. Unfortunately it suffers from $O(n^2)$ performance in the worst case, but can be expected to be $O(n \log n)$ in the average case.

After you have implemented quicksort, verify empirically that it performs no less well on random input than mergesort. To make this a little easier, you may wish to add an "accessor" to your flexarray that returns the number of elements stored in it.

The pseudocode in the textbook is on page 146 (2nd ed.) or 154 (1st ed.), but remember: their arrays start from 1 (not 0) and they pass array chunks as `(a, p, r)` where `a` is the array, `p` is the start point and `r` is the end point. In C, we prefer to pass arrays as `(a, n)` where `a` is the address of the first element we are interested in and `n` is the length of the subarray we are trying to manipulate.

Here is our pseudocode for quicksort:

```
quicksort (an array) {
  if there are less than two items in the array then stop
  let pivot hold a copy of the array's first element
  let i be an index one to the left of the array's left-most position
  let j be an index one to the right of the array's right-most position
  loop forever {
    increment i (at least once) while the value at position i < pivot
    decrement j (at least once) while the value at position j > pivot
    if i is to the left of j, then swap the values at their positions
    else break out of the loop
  }
  quicksort the left sub-array
  quicksort the right sub-array
}
```

What is really happening here?

- From the array to be sorted, we pick a value to act as a "pivot point." We just remember it, we don't shift it anywhere.
- Once we have that value, we know that if we start scanning from the left and right, we are eventually going to find a value on the left which is greater than or equal to the pivot, and a value on the right which is less than or equal to the pivot.
- Once we have found them, swap them (assuming our scanning hasn't crossed over).
- Keep doing that until the scanning does cross over.
- When finished, the right hand scan pointer (now on the left) defines for us the limit of an array where all the elements are not greater than the pivot point. Everything to the right of the pointer is not less than the pivot point. Now quicksort the two halves.

Hints

- The infinite loop in C is "`for (;;) { ... }`".
- You can break out of any loop with the statement "`break;`".
- The body of a "do while" loop is executed at least once.
- At the end of partitioning, `j` is the index of the last item on the left partition. The left partition length is therefore `j + 1` (not `j`). The right partition starts at position `j + 1`, so its length is `n - j - 1` (not `n - j`).

You should verify that your quicksort really works: it should work about as fast as (perhaps faster than) your mergesort. Fortunately it is easy to see if it is working or not, because you can recognise one of the following conditions:

1. The items don't end up sorted.
2. The program fails to terminate.
3. The sort works, but s-l-o-w-l-y.
4. The sort works, about as fast as mergesort.

When you hit condition 4, you're probably done. Don't forget to test with no input, with 1 item, and with several items all the same.

If you have time, you might try implementing a version of quicksort that either a) drops to insertion sort when an array is sufficiently small, or b) just stops sorting when an array is sufficiently small, and then runs insertion sort once over the whole array at the end.

In both cases, it is worth working out for yourself what the optimum value of "sufficiently small" should be. How much faster can you make your quicksort go?

For those who are really keen, there is a copy of Jon Bentley and Doug McIlroy's classic article, "Engineering a Sort Function" in the `$C242` directory. You should recognise in it some of the tricks we have talked about in class, and you can see how much more we *could* have talked about.

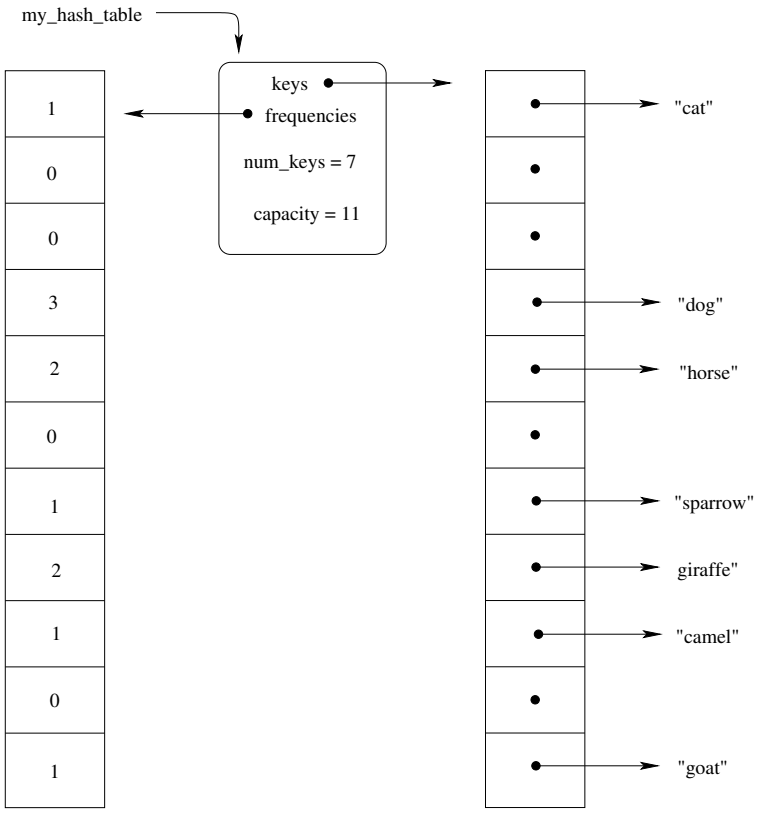
Lab 11

Hash Tables

NOTE: this laboratory is expected to take two sessions (i.e. it constitutes both lab 11 and the following lab). Material from here is used as a basis for the programming assignment.

Today we look at hash tables. Why are they special? Well, for most of their lives they have an insert time = O(1) and a search time = O(1). We can implement a hash table using an array of whatever we want to store, plus some arrays of metadata (such as how many times we've tried to store a particular item). In the real world, we would be storing key/data pairs but to make life simpler we're just going to store strings.

11.1 A Hash Table Specification



A hash table might look something like the diagram shown above. As with the flexarray ADT, our data structure is a pointer to a struct that stores useful information for us. In particular, we want to store

- How many items we can fit in the hash table (the `capacity`);
- How many items we have in the table right now (the `num_keys`);
- How many times we have repeatedly inserted each key (the frequencies: optional);
- The `keys` themselves.

The hash table in the diagram above can store 11 items, although it only has 7 right now. The key "dog" has been inserted 3 times, but "sparrow" only once.

The function prototypes for a hash table to store strings are very simple:

```
#ifndef HTABLE_H_
#define HTABLE_H_

#include <stdio.h>

typedef struct htablerec *htable;

extern void   htable_free(htable h);
extern int    htable_insert(htable h, char *str);
extern htable htable_new(int capacity);
extern void   htable_print(htable h, FILE *stream);
extern int    htable_search(htable h, char *str);

#endif
```

The `htable` type (like a `flexarray`) is a pointer to a struct. As we did with the `flexarray_new` function, we make that pointer point to something appropriate with an `htable_new` function; it takes a parameter to set the maximum number of positions in the hash table. The functions `htable_insert` and `htable_search` take keys (strings) as parameters and return the number of times that key has been entered. If the table is full, `htable_insert` should return 0 upon trying to insert a new key. Finally, `htable_print` has the job of printing out every key and frequency in the table. Rather than just assuming we want to print to `stdout`, we can pass a stream to this function, allowing us to print to `stderr` or a file instead.

Exercise

You saw in the flexarray lab how to create a struct for the contents of your flexarray. Use the diagram of the hash table to create a `struct htablerec` with the appropriate fields.

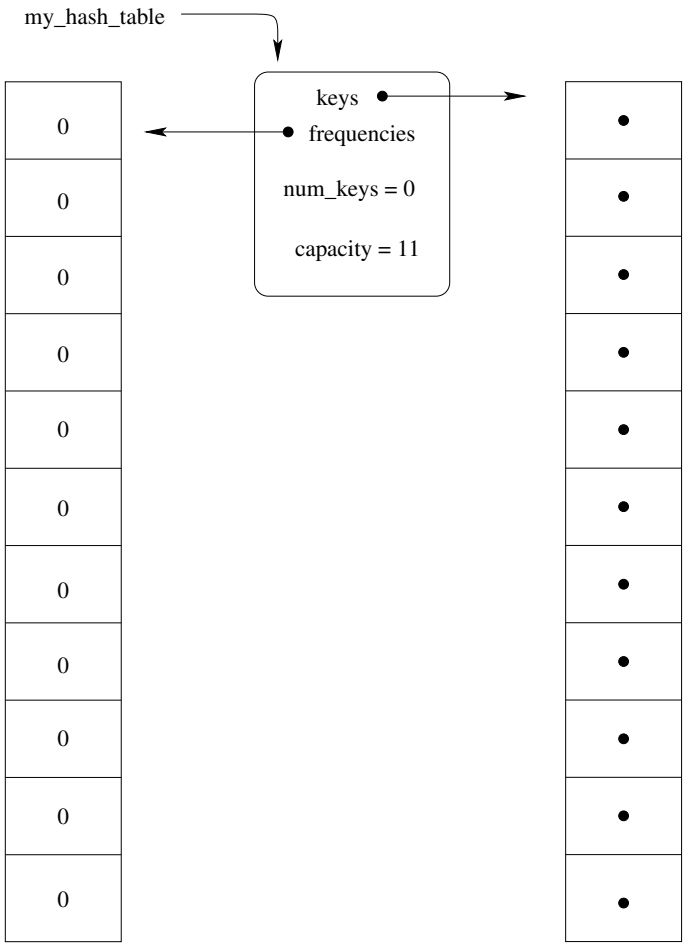
11.2 New and Free Functions

Whenever you implement a data structure you will need to write a function to create a new one as well as a function to destroy it. It is a good idea to write these functions at the same time, since whenever you allocate memory there needs to be a corresponding deallocation of that memory.

It would be good if a hash table were set up to have `NULL` character pointers in the keys array and zeroes in the frequency array. This suggests that `htable_new` should look something like this:

```
htable_new(capacity) {
  declare a htable and allocate memory as appropriate
  initialise the htable's capacity and number of keys appropriately
  allocate htable's frequency array to be able to store enough integers
  allocate htable's keys array to be able to store enough strings
  set each frequency and key to their initial values
  return the hash table
}
```

A newly created hash table of capacity 11 should look something like this:



Now let's start thinking about the `htable_free` function. Consider what would happen if you were to create a hash table called `h`, fill it up, search it for a bit, and then do this:

```
free(h);
```

The memory associated with the `struct htablerec` would indeed be freed, but how about all those allocated strings? What about the two arrays used to store keys and frequencies? That memory is not reclaimable: a memory leak has occurred. The `free` function is NOT recursive; if you wish to free things inside a pointer-based data structure, you have to go inside and free the innards systematically. You should always deallocate memory in the reverse order to which you allocated it. Don't forget that your insert function will be allocating memory for each string.

Exercise

Implement a hash table module, in files called `htable.h` and `htable.c`. Write the functions `htable_new` and `htable_free`, and create a small program to test them.

11.3 Insertion Function

Now let's look at the insertion function. In broad terms, something like this happens:

First we convert the string to an integer (possibly a very large one). Next we figure out where in the keys array the string should go. To do this we turn this newly calculated integer into an index which will fit into our hash table (which is simple: we just look at the remainder after dividing it by the capacity). This conversion of an object to an index in a hash table is called "hashing".

Once we have calculated a position in the hash table. Three possibilities occur:

1. There is no string at that position, in which case copy the string to that position and set the appropriate cell in the frequency array to 1. Increment the number of keys, and return 1.
2. The exact same string is at that position, in which case increment the frequency at that position. Return the frequency.
3. A string is at that position, but it isn't the right one. Keep moving along the array until you find either an open space or the string you are looking for. You may need to wrap around back to the beginning of the table, so each time you add to the position you should also mod by the table capacity.
   (a) If you find an open space, copy the string to that position, set the frequency to 1, and increment the number of keys.
   (b) If you find the string you were looking for, increment the frequency at that position and return the frequency.

(c) If you have kept looking for an open space but there isn't one, the hash table must be full so return 0.

Exercise

Implement the `htable_insert`, and `htable_print` functions and alter your program to check that they work properly. Use the function below to convert a string to an integer:

```
static unsigned int htable_word_to_int(char *word) {
  unsigned int result = 0;

  while (*word != '\0') {
     result = (*word++ + 31 * result);
  }
  return result;
}
```

Since it is a disaster if the key or the position it hashes to is negative, work with `unsigned int` whenever you calculate a key or a position.

When you implement `htable_print`, remember to print just the keys that exist and not the `NULL` keys. For easy formatting, it is simpler to print the frequency first and then the word.

Exercise

Use your hash table as the data structure in the following program:

```
#include <stdio.h>
#include <stdlib.h>
#include "mylib.h"
#include "htable.h"

int main(void) {
  htable h = htable_new(18143);
  char word[256];

  while (getword(word, sizeof word, stdin) != EOF) {
    htable_insert(h, word);
  }

  htable_print(h, stdout);
  htable_free(h);

  return EXIT_SUCCESS;
}
```

You'll need to add the following function (and #includes) to `mylib.c` for getting words, as well as a corresponding function prototype to `mylib.h`:

```
#include <assert.h>
#include <ctype.h>
#include <stdio.h>

int getword(char *s, int limit, FILE *stream) {
   int c;
   char *w = s;
   assert(limit > 0 && s != NULL && stream != NULL);

   /* skip to the start of the word */
   while (!isalnum(c = getc(stream)) && EOF != c)
      ;
   if (EOF == c) {
      return EOF;
   } else if (--limit > 0) { /* reduce limit by 1 to allow for the \0 */
      *w++ = tolower(c);
   }
   while (--limit > 0) {
      if (isalnum(c = getc(stream))) {
         *w++ = tolower(c);
      } else if ('\'' == c) {
         limit++;
      } else {
         break;
      }
   }
   *w = '\0';
   return w - s;
}
```

1. In the directory `$C242/labfiles` you will find the text to "War and Peace." repeated 10 times in one file. Because this is quite a large file make a symbolic link to the original rather than copying it to your current directory by using the command

   ```
   ln -s $C242/labfiles/war-and-peace-20.txt
   ```

   You can make your program count the number of words in 20 copies of "War and Peace" like this:

   ```
   ./progname < war-and-peace-20.txt
   ```

   which will give you a list (not in order) of the frequencies of each word. Pipe the output though `sort -n` to see the frequencies in order (you should find the word "the" used 692620 times).

2. Alter your program so that the desired capacity of the hash table is passed in as a command line argument. Now test the program with varying capacities of hash table (there are 18138 distinct words in "War and Peace"). You can get an approximate running time for a program by typing the command "`time`" before any command. Do you get a different time for 18143 than for 25013? Why, when "War and Peace" has fewer distinct words than either of those numbers?

In the `$C242/labfiles` directory are two very silly scripts for finding the prime number greater than (`primegt`) or less than (`primelt`) whatever argument you pass them. You may find these useful for getting prime numbers for the hash table.

11.4 Search Function

Searching a hash table is nearly the same as inserting into it, except we don't increment the frequency and we don't add in words that are not found. Our pseudocode may therefore look like this:

```
htable_search(hashtable, key) {
  create, and initialise, a value to hold the number of collisions
      we have when looking for our key
  calculate the position at which we should start looking for our key
  while there is an item at that position, but it isn't the key,
        and we haven't yet checked the entire hashtable {
      increment our position to look in the next cell
      increment the number of collisions we've had so far
  }
  if our number of collisions == the hashtable's capacity, then we can
      say that the hashtable was full but did not contain our key, so we
      should return 0
  else
      we should return the frequency at our final position
}
```

This will only work properly if the frequencies were initialised to zero and keys to `NULL` during `htable_new`.

Exercise

Add `htable_search` to your hash table implementation. Now use the hash table in the following program:

```
#include <stdio.h>
#include <stdlib.h>
#include "mylib.h"
#include "htable.h"

int main(void) {
  htable h = htable_new(113);
  char word[256];
  char op;

  /* We must have a space before the %c */
  while (2 == scanf(" %c %255s", &op, word)) {
    if ('+' == op) {
```

```
      htable_insert(h, word);
    } else if ('?' == op) {
      printf("%d %s\n", htable_search(h, word), word);
    }
  }

  htable_free(h);

  return EXIT_SUCCESS;
}
```

In the `$C242/labfiles` directory, you will find a file for testing this program. The first 50 queries should not be found (printing out a 0), and the next 50 should be (printing out a 1).

11.5 Double Hashing

Up until this point, we have been using *linear probing* as our collision resolution strategy; we just look at the next position in the event of a collision. However, we can avoid collisions more successfully if we employ *double hashing*, which involves calculating a step based on the integer you get from the converted string.

In the following exercise, work on a **copy** of your hash table implementation.

Exercise

1. Add a static function `htable_step` to your `htable.c` file. It looks like this:

```
static unsigned int htable_step(htable h, unsigned int i_key) {
  return 1 + (i_key % (h->capacity - 1));
}
```

2. Alter your `htable_insert` routine so that it calculates a "step" before beginning to search the keys array. Then, instead of adding 1 to the position each time a collision occurs, add the step instead.

3. Alter your `htable_search` routine similarly.

Now use the altered hash table to do the word frequency counting exercise on page 75. Use "War and Peace" again as your test file. Time your original hash table doing the job and your new "double-hashing" hash table as well. Does there appear to be a difference? What about if you alter the hash table capacities?

Lab 13

Practical Test II

13.1 Practical Test: Tuesday August 18^{th}

During this lab session you will be sitting a practical test. Full descriptions of the tests will be available on the course web page:

`http://www.cs.otago.ac.nz/cosc242/assessment.php`

Lab 14

Binary Search Trees

NOTE: this laboratory is expected to take two sessions (i.e. it constitutes both lab 14 and the following lab).

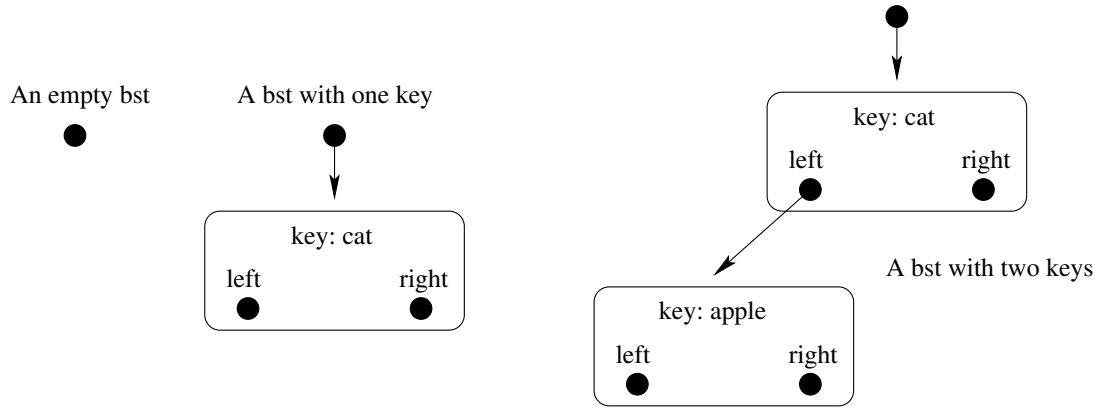
You will also need to put in some preparation time and post mortem time since this can be a tricky topic. Material from here is used as a basis for the red-black trees in later labs which form part of the programming assignment.

Binary Search Trees (BSTs) give us a container type with properties that differ from hash tables. Search and insert routines are O(log n) rather than O(1), but space utilisation is proportional to the number of items stored rather than the number of storage slots.

BSTs also give us an opportunity to explore a recursive data structure. Rather than "wrapping up" the internals of the ADT with a record that stores statistics regarding it, we're going to simplify a little: our `bst` will be a pointer to a node which itself has two `bst`s as data fields: one for the left subtree and one for the right. This way every major routine can be described and implemented recursively. This is not the way the textbook does it, but here we have an opportunity to compare side by side a recursive description/implementation (ours) with an iterative one (the textbook's).

14.1 Defining and Searching

We can describe a `bst` as being either `NULL`, or a pointer to a struct with fields `key` (a string), `left` (a `bst`), and `right` (a `bst`). We don't need parent pointers or sentinels or any other fancy stuff: this simple definition gives us all we require:



So our definition of a `bst_node` is nice and simple:

```
typedef struct bst_node *bst; /* should live in bst.h */

struct bst_node { /* should live in bst.c */
  char *key;
  bst left;
  bst right;
};
```

Given a recursive definition of the tree, searching it is an elegant and easy-to-describe operation. Let's assume that if the desired string is found, 1 is returned to the caller; otherwise 0 is returned. Our function header would look like this:

```
int bst_search(bst b, char *key) { ...
```

Inside our search function we need to consider a number of cases. If we're looking at a `NULL` tree (or subtree), it's empty so we aren't going to find our key here. We can return 0 safe in the knowledge that the key does not exist in this tree. Conveniently, this is one of our stopping cases—all nicely taken care of. If we are looking at a node and its key is the same as the one we are looking for, we can return 1. If the node's key is too big, return the result of searching the left subtree. If the node's key is too small, return the result of searching the right subtree.

14.2 Inserting

Warning: inserting into a recursively defined bst is fundamentally different in style to inserting into the ADTs you have seen so far. Why? Well, consider this example:

```
bst b = NULL; /* start with empty tree */
bst_insert(b, "foo"); /* WILL NOT WORK */
```

The problem is that `b` is a pointer whose value is set to `NULL` and in C all parameters are "pass-by-value"—their values remain unchanged after the function finishes. This means

that no matter what we do to `b` inside `bst_insert`, it will still be `NULL` after `bst_insert` ends.

The solution is presented in Kernighan and Ritchie in Chapter 6: return the new tree from the insert function. Instead of the code above, you would do this:

```
bst b = NULL; /* start with empty tree */
b = bst_insert(b, "foo"); /* possibly assigns b a new address to store */
```

This means we have to specify the `bst_insert` function very carefully. In particular, we need to make sure that we return `b` at the end of our function so that the reassignment can happen.

Here is an overview of the cases to consider.

If we're trying to insert into an empty tree, we can allocate memory, copy in the key, and return the result. Otherwise a key must already live here. If it's the same, do nothing (no duplicates). If the key to be inserted is smaller, then we should insert into the left subtree. If the key to be inserted is greater, we should insert into the right subtree. In either case, note that we ASSIGN the result of the insert to the appropriate subtree. In the main program, an insert will always look like this: `b = bst_insert(b, "string");` so when we insert into a subtree, we have to do the same thing:

```
b->left = bst_insert(b->left, "string");
```

Incidentally, is this "assignment of the result of inserting" so terribly strange? Not really. Consider how we might implement a function whose job was to double the value of a variable. Its behaviour is based on whatever state the variable is in now, just as the behaviour of `bst_insert` is dependent on the current state of the tree. We might call the "doubler" function like this: `i = doubler(i)`, and it would make perfect sense. What if we wanted a function to add some number `j` to our variable? We'd call it like this: `i = add(i, j)`. It's the same with our tree: we want to create a new state of the tree based on the old state plus some new information, giving us a call of the same form:

```
b = bst_insert(b, "new information");
```

14.3 Iterating and Traversing

When we've wanted to print out an ADT such as a list, we've just created an `ADT_print` function which visits every item and copies it to the standard output. But consider these points:

- There are other things we may like to do that involve visiting every item in an ADT (perhaps in some special order) such as systematically updating all the data parts of key/data pairs.
- Whatever we want to do to each item, the code for visiting them all remains the same. Why should we repeat it? Why don't we just write an `ADT_iterate` function which takes two parameters: an ADT to work on, and a *procedure* to perform at each item?

Fortunately, C lets us do just that, because C lets us pass function names as parameters to other functions. It looks something like this:

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

void foo() {
  printf("Doing foo\n");
}

void bar() {
  printf("Doing bar\n");
}

void baz(void f()) {
  f();
}

int main(void) {
  baz(foo);
  baz(bar);
  return EXIT_SUCCESS;
}
```

The function `baz` is declared as requiring the name of a function (one which returns nothing and takes no parameters itself). We write two such functions (`foo` and `bar`). The job `baz` does is very simple: it just calls the function it was passed.

Consider what this capability means for us—we can write an `ADT_iterate` function with two parameters: an `ADT` and a function which operates on either keys or satellite data. The job of `ADT_iterate` is to visit every piece of key or data in the ADT and call the desired function on it.

For a list, there are two iteration orders that we might be interested in: from front to back, and from back to front. For trees, there are two *types* of iteration order that we are interested in: one which hits every key in key order, and one which visits every node in such a way that the "shape" of the tree is exposed. The first is called an "inorder" traverse and looks like this:

```
bst_inorder(bst b, void f(char *s)) {
  if b is NULL then return /* stopping case */
  inorder traverse the left subtree
  apply f to b->key
  inorder traverse the right subtree
}
```

The other traverse that we are particularly interested in is called "preorder":

```
bst_preorder(bst b, void f(char *s)) {
  if b is NULL then return /* stopping case */
  apply f to b->key
```

```
  preorder traverse the left subtree
  preorder traverse the right subtree
}
```

If you print out a tree using a preorder traverse, it is easy to visualise the actual structure of the tree. Try it on a piece of paper: draw a tree, list the keys in preorder, then swap the list with a friend. You should be able to regenerate exactly the same tree that your friend started with, and your friend should get your tree.

If we put the function-application part at the end of the traversal calls, we'd get a "post-order" traverse. Can you think of anything you might want to do which would make a post-order traversal useful? How about freeing memory when disposing of a tree?

Exercise

Create and test a binary search tree ADT. Here is the header file `bst.h`:

```
#ifndef BST_H_
#define BST_H_

typedef struct bstnode *bst;

extern bst   bst_delete(bst b, char *str);
extern bst   bst_free(bst b);
extern void  bst_inorder(bst b, void f(char *str));
extern bst   bst_insert(bst b, char *str);
extern bst   bst_new();
extern void  bst_preorder(bst b, void f(char *str));
extern int   bst_search(bst b, char *str);

#endif
```

You should create a `bst.c` file in the usual way, and implement each of the functions (except free, until you have read the next section). The only one we haven't really considered is `bst_new()` which gets a `bst` into its starting state: i.e. it does nothing except return `NULL`.

Once you have done this write a program to help you test your bst. You could expand on the example used to test `htable_search` in the previous lab. For the function which you pass to `bst_inorder` and `bst_preorder` you could use something like this:

```
void print_key(char *s) {
  printf("%s\n", s);
}
```

14.4 Deletion

You may have noted that the `bst.h` file contains a prototype for deleting items from a binary search tree. It needs to return a `bst` (like insertion) because when we delete from a node we

will set it to `NULL`. The only way we can do that (if it is currently pointing to a particular memory address) is to return the new value; hence a call to `bst_delete` will look like this: `b = bst_delete(b, "some value")`.

We are doing several things differently to the textbook here:

1. We are deleting by *key value* rather than by node address. This is so that our main program doesn't have to know anything about what a `bst_node` looks like, but it also means that we have to search for the key first.
2. Our delete function isn't going to depend on the *successor* function, as the one in the textbook does. However, we are going to use a nearly identical strategy to find a replacement node in the case of deleting a node with two children.
3. Our delete is going to be defined recursively, just like all our other `bst` functions.

Here is a recursive description of deleting a particular key from a BST:

- Deleting a key which doesn't exist in the tree should have no effect.
- If the key you wish to delete matches the key at the current node, splice out the node.
- If the key is smaller than the key at the current node, delete the key from the left subtree.
- If the key is larger than the key at the current node, delete the key from the right subtree.

Splicing out the node is easy in two cases and a bit tricky in the third. Here are the three cases:

- If the node to be deleted is a leaf: free the key, free the node, set the node's pointer to `NULL`.
- If the node to be deleted has one child: make the pointer point to the child instead. Free the key and free the node.
- If the node has two children: find the leftmost child of the right subtree (the successor). Swap the key with the successor's key. Now delete the key from the right subtree.

It is important to code this up slowly and carefully; it is easy to make a misstep here (especially since we are using strings as keys—that's an extra bit of memory manipulation to worry about).

Exercise

Add the implementation of `bst_delete` to `bst.c`. Add testing code to make sure all cases of deletion work correctly.

Lab 16

Red-Black Trees

NOTE: this laboratory is expected to take two sessions (i.e. it constitutes both lab 16 and the following lab).

You will also need to put in some preparation time and post mortem time because this can be a tricky topic. Material from here will form part of the programming assignment.

Unfortunately we can rarely guarantee a near-optimal order of insertion for tree structures, meaning our trees can be badly balanced and our search time closer to O(n) rather than O(log n). One way to ensure O(log n) search is to perform *rotations* on subtrees in order to keep them approximately balanced. Red-black trees are approximately balanced trees that ensure no subtree is more than twice the height of its sibling.

16.1 Colours

Red-black trees need to have a colour associated with every node (`NULL` nodes are black). We also need a way of quickly deciding whether a node is black or red, preferably without incurring the overhead of a function call.

Storing the colour is easy enough: We simply declare an enumerated type with values `BLACK` and `RED`, then typedef that to create a `rb_colour` type. Now a tree node can have an extra field of type `rb_colour`, which may take on the values `RED` or `BLACK`.

```
typedef enum { RED, BLACK } rbt_colour;

struct rbt_node {
  char *key;
  rbt_colour colour;
  rbt left;
  rbt right;
};
```

To decide if a node is black, we can add the following macro:

```
#define IS_BLACK(x) ((NULL == (x)) || (BLACK == (x)->colour))
```

and for red we add the macro:

```
#define IS_RED(x) ((NULL != (x)) && (RED == (x)->colour))
```

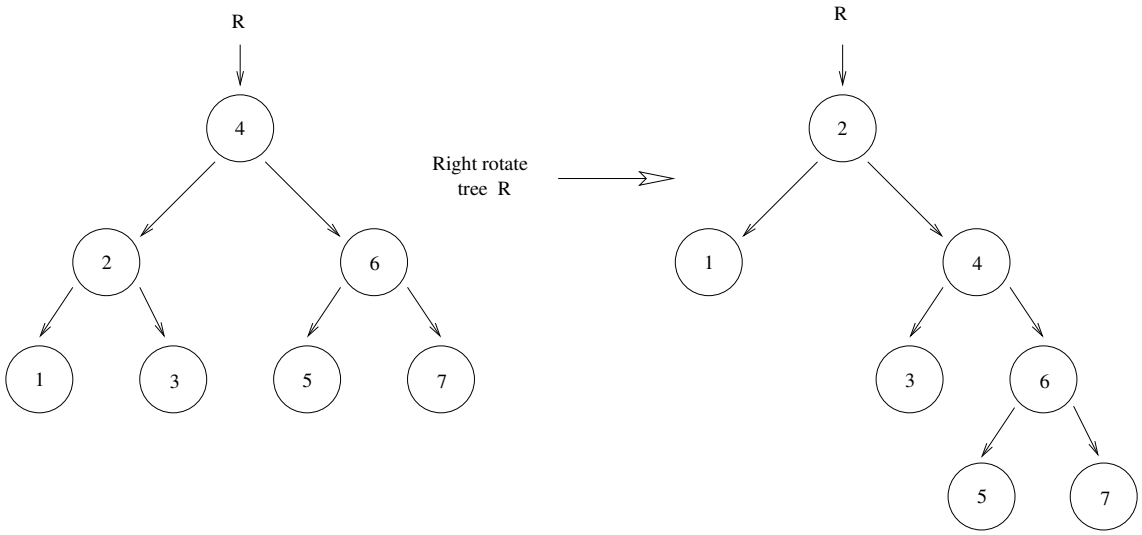
Exercise

Make copies of `bst.h` and `bst.c` and call them `rbt.h` and `rbt.c`. Change every instance of `bst` to `rbt` in both files. Add the definitions above to `rbt.c`. Make sure that nodes are coloured red upon creation. Compile and test so as to verify that your RBT still works and behaves exactly as a BST.

16.2 Rotations

As we emerge from the recursion of inserting a new (red) node, we're going to check (by looking *downwards*) that we haven't violated any red-black properties. In fact, the only one that we can violate is the one that says "if your colour is red, you shall not have a red child". When we see this occur, we'll "fix up" the situation by recolouring nodes and *rotating* branches if necessary. The idea is that the things we do to fix up the colour violation will never ruin the property that says "the black-height of every path to a leaf shall be the same".

We will need two types of rotation: left-rotation (moving branches from the right to the left) and right-rotation (moving branches from the left to the right). The result of a right-rotation of a tree with root *R* looks like this:

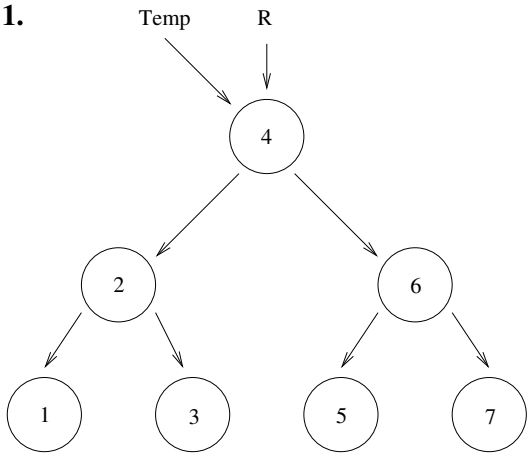


Since we could be changing the address stored by the main pointer of the subtree, `right_rotate` needs to return the new value (just as we did in `bst_insert`). So a call to `right_rotate` would look something like:

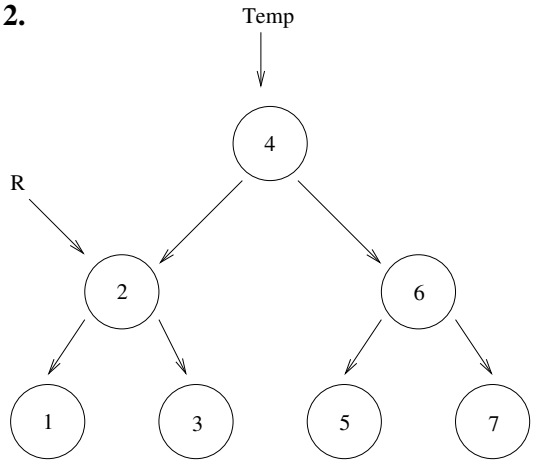
```
r = right_rotate(r)
```

Here are a few pictures which show the steps we need to go through to perform a rotation. Each step corresponds to one line of code in our function.

1.

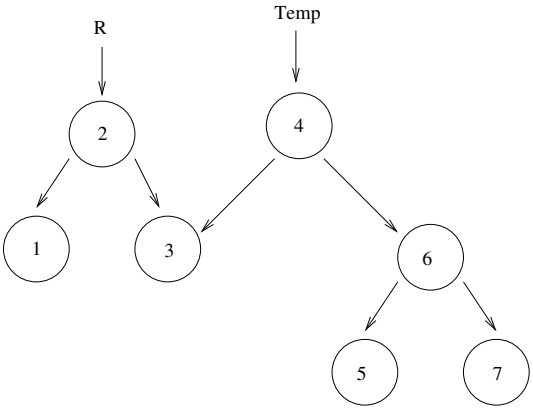


2.

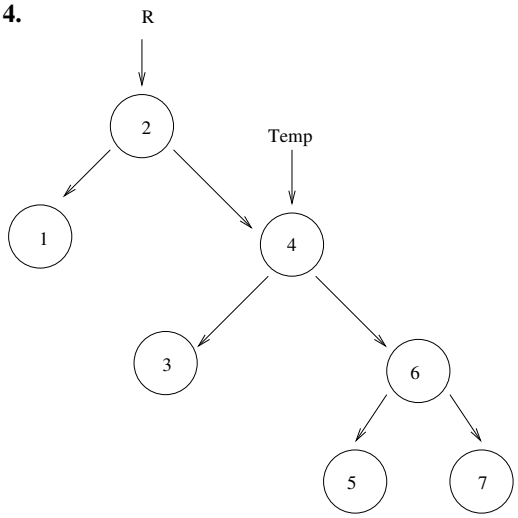


First we need to keep track of the original root of the tree. Then we change the root of the tree to point to its left child.

3.



4.



Next we make the left-child of temp (the original root of the tree) point to the right child of the new root of the tree. Lastly we make the right child of the new root of the tree point to the old root of the tree (temp).

Don't forget that we need to return the modified tree at the end of our rotate function.

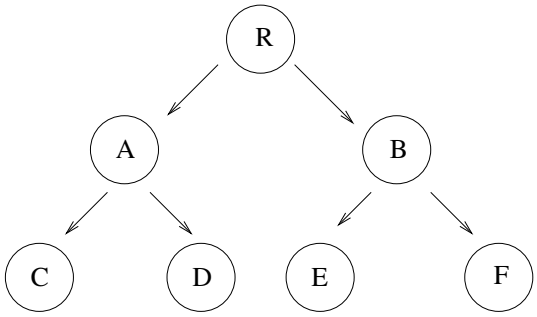
Once you have written `right_rotate` it is easy to write `left_rotate`. It is almost exactly the same, except that every instance of "left" gets replaced with "right" and vice versa.

16.3 Insertion

Insertion into an RBT is almost exactly the same as insertion into a BST. We can reuse the code from `bst_insert` with the only changes being:

- Every instance of bst is now rbt.
- We initialise the colour field of our struct.
- We call `rbt_fix` before returning `r`.

As the recursion of our `rbt_insert` unwinds, `rbt_fix` looks *downward* through the nodes to see if any children/grandchildren pairs are both red. If this happens, rotations and recolourings are performed (depending on the colour of the child on the other side). In the tree below, when an `rbt_fix` is performed on the root (*R*), we look down to see if any one of the pairs A and C, A and D, B and E, or B and F are red. If they are then recolourings and rotations are performed according to table shown on page 88



It is a good idea to draw a few pictures to make sure you understand what happens when a rotation occurs. If you are unsure of anything then come and have a chat with a demonstrator to clarify things.

Here is a description of the changes that happen when we perform a left rotation on *A*:

- *D* becomes *R*'s left child.
- *A* becomes *D*'s left child.
- Whatever had been *D*'s left child becomes *A*'s right child.

Your lecture notes, as well as the textbook, contain a number of helpful examples of the steps which occur when balancing a red-black tree.

Below are two tables showing the steps which need to be taken to 'fix' a tree when an insertion has resulted in two consecutive reds.

Consecutive reds below R	Other child	Action
left child (A), A's left child (C)	B is red	colour root (R) red and children (A,B) black
	B is black	right-rotate root (R), colour new root (A) black, and new child (R) red
left child (A), A's right child (D)	B is red	colour root (R) red and children (A,B) black
	B is black	left-rotate left child (A) , right-rotate root (R), colour root (D) black and new child (R) red

Consecutive reds below R	Other child	Action
right child (B), B's left child (E)	A is red	colour root (R) red and children (A,B) black
	A is black	right-rotate right child (B), left-rotate root (R), colour root (E) black and new child (R) red
right child (B), B's right child (F)	A is red	colour root (R) red and children (A,B) black
	A is black	left-rotate root (R), colour root (B) black and new child (R) red

Exercise

First add static methods for `left_rotate` and `right_rotate` to your `rbt.c` file. Now write pseudocode for `rbt_fix` that will take care of all of the cases above. Next add a static method `rbt_fix` to `rbt.c` and adjust `rbt_insert` accordingly. Write a new pre-order traversal that prints out whether the node is red or black. (Note that you can't use a regular iterator to do this, since the function passed to the iterator can only see the key and not the internal node structure of the tree.) Now test the tree with input

```
"h", "l", "i", "f", "j", "g", "k", "d", "a", "e", "b", "c"
```

(the second string is a lower-case L). The interesting thing about this permutation of strings is that it exercises *every* case in `rbt_fix`. On a pre-order traverse, you should get:

```
black: g
red:   d
black: b
red:   a
red:   c
black: f
red:   e
red:   i
black: h
black: k
red:   j
red:   l
```

REMEMBER, any function which could alter the root of a tree must return the modified tree so that it can get reassigned using a statement like this:

```
r = rbt_fix(r)
```

Lab 18

Two Queue Implementations

Today we will be implementing two different versions of a *queue* data type. Queues are very commonly used in computer programs and have two basic operations, *enqueue* and *dequeue*, which provide first-in first-out (FIFO) behaviour. A real-world equivalent could be be an escalator where no-one overtakes anybody or goes the wrong way.

18.1 An Array Based Queue

Our first implementation will use an array to hold items which are added to the queue. We could decide that index 0 in the array would always refer to the front of the queue, but this would lead to a lot of time spent shuffling items along by one position whenever anything is removed from the head of the queue. To get around this problem we can treat the array as a circular buffer. This means that the index of the head of the queue gets incremented whenever remove an item. If the head (or tail) of the queue ever reaches the last index of the array the next dequeue (or enqueue) will access the first index in the array. This can be accomplished by dividing by the size of the array, and using the remainer as the index.

If we were going to use our queue to store doubles then we could define our queue like this:

```
struct queue {
   double *items;
   int head;
   int capacity;
   int num_items;
};
```

To enqueue an item, we add it at the position `(head + num_items) % capacity` and then increment `num_items`.

To dequeue an item we just return the contents of `items[head]` and then update `head` and `num_items` appropriatedly.

Before adding an item to the queue you should check that `num_items` is less than the `capacity`.

Before removing an item from the queue you should check that `num_items` is greater than 0.

Exercise

Create and test an array-based circular queue. Check that it behaves as it should when both the head and tail of the queue wrap around. It's best if you give the queue a small default size, say 5, to make testing it easier. Here is the header file `queue.h`.

```
#ifndef QUEUE_H_
#define QUEUE_H_

typedef struct queue *queue;

extern queue queue_new();
extern void enqueue(queue q, double item);
extern double dequeue(queue q);
extern void queue_print(queue q);
extern int queue_size(queue q);
extern queue queue_free(queue q);

#endif
```

The `queue_new` and `queue_free` functions allocate and deallocate memory for the queue, while `queue_size` returns the number of items in the queue. The `queue_print` function prints out items in the queue, one per line, from head to tail. Keep the code that creates and tests your queue in a separate file, say "queue-test.c".

18.2 A Linked List Queue

A major drawback with using a circular array to hold a queue is that the queue will have a fixed maximum size. If the queue gets too full then it would not only require resizing, but also all of the items need to be copied into their new positions. For some applications having a fixed maximum capacity is fine. We shall see this when performing a breadth first search on a graph in a later lab.

Implementing our queue as a linked list avoids the fixed size problem. It is a little more complex than an array based implementation, but after having worked with BSTs and RBTs you should find it relatively simple.

To get you started here is a definition of a linked-list based queue:

```
typedef struct q_item *q_item;

struct q_item {
   double item;
   q_item next;
};

struct queue {
   q_item first;
   q_item last;
   int length;
};
```

Your `queue.h` should remain unchanged since we are only changing the underlying implementation not the interface itself.

Exercise

Create a linked list based queue using the definition above in a file called `queue2.c`. You should be able to compile and run it using the `queue-test.c` that you created previously. Make sure that it behaves the same as the array based version except for the fixed size limitation.

Make sure that you get both of these implementations working correctly because you will need to have a working queue for the first graph lab next week. They are also used in the upcoming third practical test.

Lab 19

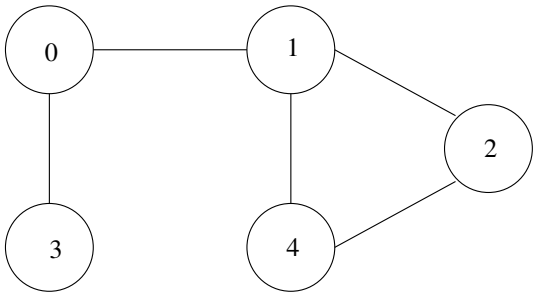
Graphs 1

19.1 Graph Representation

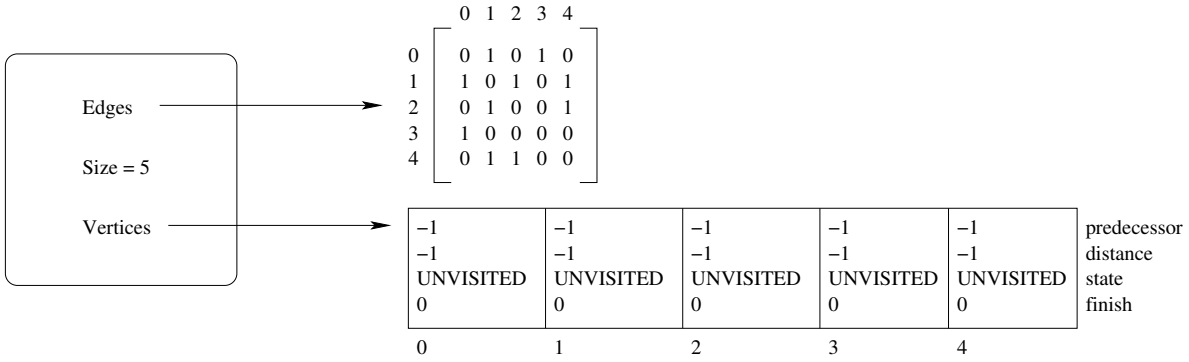
Let's consider first the things we might like to be able to do with graphs: breadth-first search (and perhaps shortest paths), and depth-first search (and perhaps strongly-connected components). We therefore need at least the following:

- A graph type of which we can create an arbitrary number of instances (as we could with hash tables, trees etc.).
- Functions which take a graph as the first argument (and possibly vertex numbers as subsequent arguments) which perform the algorithms we're interested in.
- A representation that captures the data needed for those algorithms.

Here is a diagram of a graph that would meet the basic requirements of breadth-first and depth-first search:



And here is a possible representation.



This sort of thing should look pretty familiar by now. We have a graph of 5 vertices and 10 edges (since it is undirected). The vertices are labelled from 0 to make the C representation easier. Our graph type can be the usual pointer to a struct containing the size, the adjacency matrix, and an array of information regarding each vertex (with vertex 0's data stored in position 0, 1's in position 1, and so forth).

Since we've done a lot of this before, let's see if you can draw together all the techniques you have been presented with over the semester:

Exercise

Create `graph.h` and `graph.c` files. Write a test program that lets you create a graph and print it out in some sensible fashion.

Hints:

- Follow a similar pattern to `htable`; i.e. create a graph type that is a pointer to a graph record, and define `struct graphrec` in `graph.c`. Write a `graph_new` routine that returns an initialised graph, using `emalloc` to allocate memory for the graph struct, the edges matrix, and the vertices array. The `graph_new` routine should take an integer argument to say how many vertices the graph has.
- There are several ways you could do the vertices array. You could either follow the `htable` pattern and keep separate arrays for state, predecessor, etc. or you could create a `struct vertex` and then make a single array of those using `emalloc`. If you choose the latter (and define a vertex to be a struct rather than a pointer to a struct), you can access the current state for vertex 2 from a graph g with

  ```
  g->vertices[2].state
  ```

  whereas for the first technique you would use

  ```
  g->state[2]
  ```

- You can create a `state_t` type with the members `UNVISITED`, `VISITED_SELF`, and `VISITED_DESCENDANTS` in the same manner as you did for red-black trees.
- Initialising the adjacency matrix can be a little tricky. Think carefully about what is going on when you do this. A matrix is an array of arrays, so the `edges` field of your graph should have type `int **`. You have to initialise the number of rows first, and then initialise each individual row. If you are unsure how to do this, come and have a talk to one of the demonstrators.
- The `graph_new` routine should return a graph with no edges. How do you connect up the edges? Obviously you will need a `graph_add_edge` function. What should its arguments be? It would be nice to have one routine that just adds an edge from vertices u to v (for directed graphs) and a different function to add bidirectional edges (i.e. one from u to v and another from v to u for undirected graphs). Then you can use the same graph routines for directed or undirected graphs.
- Don't forget about `graph_free` function. It is always a good idea to write your free function straight after new.
- You may like to print your graphs so that you can see exactly what is being stored after creating a graph and adding edges. How about printing it as if it were stored as an adjacency list (even though it isn't)?

19.2 Breadth-first Search

In an unweighted graph, doing a breadth-first search will allow you to store a shortest path from a source vertex to all other vertices. The textbook method of doing this is to colour the vertices to distinguish between undiscovered (white), discovered (grey), and done (black). To make things easier to understand we are using `UNVISITED`, `VISITED_SELF`, and `VISITED_DESCENDANTS` instead of white, grey, and black.

We use a queue of integers to store which vertices we are going to explore next, and also to determine the order that we will explore them in.

Exercise

Implement the breadth-first search algorithm detailed on page 532 of the textbook (470 in 1st edition).

```
graph_bfs(graph, source) { /* source = vertex we find distances from */
  for each vertex in the graph {
    set state as unvisited
    set distance to infinity (use either INT_MAX from limits.h or -1)
    set predecessor as non-existent (predecessors start at 0 so use -1)
  }
  set source's state to visited_self
  set source's distance as 0
  add source to the queue
  while queue is not empty {
```

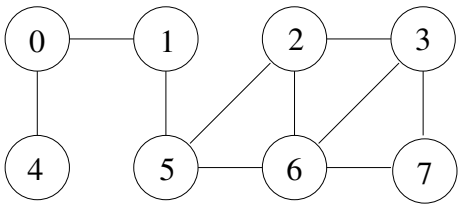
```
    set vertex u to be a vertex removed from queue
    for each vertex v adjacent to u, that has not been visited {
      set its state to process of being visited
      set its distance to be 1 + the distance to u
      set its predecessor to u
      add v to queue
    }
    set u's state to visited_descendants
  }
}
```

We shall not worry about selecting additional sources in order to ensure that every vertex, even in a disconnected graph, is visited. We'll just visit every node that we can get to via BFS starting from node *source*.

You should be able to use one of your queue implementations from lab 18 modified to store integers instead of strings. If you use the array based version you can initialise it to the size of the graph, since there are never more items stored in it than there are vertices in the graph.

Test your `graph_bfs` routine by creating the graph pictured below, doing breadth-first search on it starting from node 1, then printing out the graph so that the distances of each vertex are visible. A step-by-step example of breadth-first search on this graph (with vertices named r, s, ..., y instead of 0, 1, ..., 7) can be found on page 533 of the 2^{nd} edition of the textbook.

The textbook shows the final distances in part (i); your distances should match those.



Your output for this exercise should look like this:

```
adjacency list:
0 | 1, 4
1 | 0, 5
2 | 3, 5, 6
3 | 2, 6, 7
4 | 0
5 | 1, 2, 6
6 | 2, 3, 5, 7
7 | 3, 6

vertex distance pred
    0       1    1
    1       0   -1
    2       2    5
    3       3    2
    4       2    0
```

```
5       1   1
6       2   5
7       3   6
```

You should initialise the graph in `main()` by using a sequence of calls to your `graph_add_edge` function (or perhaps your `graph_add_2edges` function) with numbers read from *stdin*. You should test your program with the input files from the `$C242/labfiles/graph-input` directory. The first number in each file determines how many nodes a graph contains. Each subsequent pair of numbers represents an edge to be added.

Lab 20

Graphs 2

20.1 Depth-first Search

Unlike breadth-first search, depth-first search will not require you to implement another data structure as part of the solution. Instead of using a queue to process the vertices, depth-first search uses a stack—and the easiest stack to use in C is the system stack in recursive calls. The textbook does it that way too. Here is some pseudocode which describes what we want to do:

```
graph_dfs(graph) {
  for each vertex in the graph {
    set its state to unvisited
    set its predecessor as non-existent (use -1)
  }
  initialise the step to 0
  for each vertex v in graph {
    if its state is unvisited {
      visit(v)
    }
  }
}

visit(vertex v) {
  set v's state as visited_self
  increment step
  set distance to v as step
  for each vertex u adjacent to v {
    if it is unvisited {
      set its predecessor as v
      visit(u)
    }
  }
  increment step
  set v's state as visited_descendants
  set v's finish value to step
```

```
}
```

This time, we shall select additional source vertices so as to process the whole graph, even if disconnected. The trick is to have a non-recursive part of the algorithm which checks for unvisited vertices and, upon finding one, calls the recursive visiting algorithm.

The other unusual thing here is the global variable `step`. The easiest way to maintain it is as a `static int` variable with file scope, so that it exists outside the recursion stack.

Exercise

Implement depth-first search and test it by creating and searching the graph on page 542 of the textbook (described by the adjacency list below). When you print the graph, print the distance and finish fields as well: they should match those given below.

You should adjust your graph printing function so that it includes the "finish" information as well as distance and predecessor:

```
adjacency list:
0 | 1, 3
1 | 4
2 | 4, 5
3 | 1
4 | 3
5 | 5

vertex distance pred finish
   0       1   -1    8
   1       2    0    7
   2       9   -1   12
   3       4    4    5
   4       3    1    6
   5      10    2   11
```

Task: Please draw the graph that is represented by the adjacency list above. Remember to make the edges directed.

Lab 21

Extension graph exercises

A Small Challenge

It is quite possible that you will finish the previous graph material which will be assessed in the third practical test sooner than expected. If so, you may be interested in trying these extensions to BFS and DFS.

Create a `graph_print_shortest_path` routine that takes a graph and two vertex numbers as arguments. The routine should use the predecessor fields of the graph after a breadth-first search to print out a shortest path between the first vertex argument and the second vertex argument.

A Bigger Challenge

Once you have done depth-first search, implement `graph_print_scc` (which should print the strongly connected components of the graph). You will need to add a routine that calculates and stores sensibly the transpose of the edges matrix (i.e. every edge is made to point in the opposite direction), because the algorithm looks like this:

```
strongly_connected_components(graph g)
  perform a dfs with g to compute finishing times for each vertex
  calculate the transpose of g
  call dfs on the transpose of g, but in the main loop of dfs process
       the vertices in order of decreasing finishing time
  output the vertices of each tree in the depth-first forest of the
       previous step as a separate strongly connected component
```

Hint: in order to achieve the last step, consider modifying `dfs_visit`, which processes "clusters" (elements of a forest) by recursion.

Test `graph_print_scc` by creating the graph on page 553 (2nd ed., page 489 in 1st ed.), replacing a, b, c... with 0, 1, 2....

Lab 22

Practical Test III

22.1 Practical Test: Friday Sept 25^{th} or Tuesday Sept 29^{th}

During one of the above lab sessions you will be sitting a practical test. Full descriptions of the tests will be available on the course web page:

`http://www.cs.otago.ac.nz/cosc242/assessment.php`.

Lab 23

Dynamic Programming

In our final lab we will look at dynamic programming. The specific example that we will use was covered in lectures and is dealt with at length in the textbook on pages 350-355 (2^{nd} edition).

23.1 Longest Common Subsequence

A subsequence is defined in the textbook like this: "A subsequence of a given sequence is just the given sequence with some elements (possibly none) left out." For example, "brown", "bow", "bn" and "w" are subsequences of the sequence "brown", while "now", "wo" and "bozo" are not.

Exercise

Start by writing a program which takes two arguments from the command line. Write a recursive function to calculate the **length** of the longest common subsequence of those two arguments.

Here is the recursive definition of how to do this from the textbook:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

where $c[i, j]$ is the *length* of an LCS of the sequences X_i and Y_j, and the goal is to find the length of an LCS of X, Y (i.e. the sequences X_m and Y_n where $m = len(X)$ and $n = len(Y)$).

Try testing your program on the following input:

```
abcbdab             bdcaba
abcdefghijklmno     onmlkjihgfedcba
abcdefghijklmnop    ponmlkjihgfedcba
abcdefghijklmnopq   qponmlkjihgfedcba
```

You should notice that the time taken to run your program is increasing exponentially (as you would expect). Now try implementing the `lcs_length` algorithm detailed on page 353 of the textbook (2^{nd} edition). To make things simpler we won't worry about generating the b table (the one containing the arrows). This is how we can fill the c table with numbers in order to calculate the length of the lcs much more efficiently than our recursive method:

```
lcs_length(table, seq1, seq2) {
   initialise column 0 of table to 0
   initialise row 0 of table to 0
   for i = 1 to length of seq1 {
      for j = 1 to length of seq2 {
         if characters at position i-1 of seq1 and j-1 of seq2 match then
            set value at table[i,j] to value of table[i-1,j-1] + 1
         else if table[i-1,j] >= table[i,j-1] then
            set value at table[i,j] to value of table[i-1,j]
         else
            set value at table[i,j] to value of table[i,j-1]
      }
   }
}
```

You might find the function below useful. It prints out a table like the one on page 354 of the textbook (2^{nd} edition). This will help you to check that your `lcs_length` function is working correctly. It prints out a table containing the lcs at any given point. The sequences used are printed out down the side and across the top.

```
void print_table(char *seq1, char *seq2, int **table, int rows, int cols) {
   int i = 0, j = 0;
   int width = table[rows][cols] > 9 ? 2 : 1;
   printf("%*c ", width+1, ' ');
   while (seq2[i] != '\0') {
      printf("%*c", width+1, seq2[i++]);
   }
   printf("\n");
   for (i = 0; i <= rows; i++) {
      for (j = 0; j <= cols; j++) {
         if (j == 0) {
            printf("%c ", (i==0) ? ' ' : seq1[i-1]);
         }
         printf("%*d ", width, (table[i][j]));
      }
      printf("\n");
   }
}
```

When you have successfully implemented the `lcs_length` function you can find the length of the lcs at the bottom right-hand corner of the table. As a final exercise write an `lcs_print` function which starts at the bottom right hand corner of the table and recursively works back through the table to print out the lcs itself. If you want more of a challenge then don't look at the pseudocode given at the bottom of this page.

```
lcs_print(table, seq1, row, col) {
   if row or col is 0 return
   if table[row,col] > table[row-1,col] and table[row,col-1] then
      call lcs_print with row-1 and col-1
      print the character at seq1[row-1]
   else if table[row,col] > table[row-1,col] then
      call lcs_print with row and col-1
   else
      call lcs_print with row-1 and col
}
```

Appendix A

Writing and testing code

During this course you will be writing lots of programs of various sizes. This appendix is designed to help with the task of creating these programs, and ensuring that they work correctly.

Defining the task

Before we can write a program we need to know what it is supposed to do. For a very basic program a sentence or two should suffice. Let's say we are going to write a program which performs the same task as the first practical test in lab 7 (but without some of the extra requirements - such as using a recursive print function). A simple outline could be:

```
Read in a number of words from stdin and calculate the average length.
Print out the average length, followed by a list of all of the words
which are longer than average.
```

We should think about things like:

- What defines a word?
- How many words should we be able to handle?
- What if more than the maximum number of words is given?
- What if no words at all are given?
- How long can a word be?
- What if the length of a word is exceeded?

Let's say we came up with the following answers:

- A word is any sequence of non-white-space characters.
- We want to handle a maximum of 100 words.

- Any extra words are ignored.
- If no words are input there should be no output since there would be no average length (and no words longer than average).
- A word must not exceed 79 characters.
- Words exceeding 79 characters will be truncated with the remaining characters starting the next word.

After answering these questions we can draw up a pseudocode outline of our program.

```
allocate space for our words

while we haven't exceeded our word maximum and there is another word {
  read the next word
  add it to our list of words
  increment our word counter
}

if we actually read any words {
  calculate the average length
  print the average length
  print longer than average words
}
```

Since this is just pseudocode we can choose how detailed we want to be at this point. In the above example we included some low level things like *allocate space for our words* and *increment our word counter*, but the operations *calculate the average length* and *print longer than average words* were written as single statements even though they must involve a loop of some sort.

Writing the code

At this point we could open up an editor and write the code for our whole program in one go. However, since it is quite likely that we will make a number of mistakes, a better approach might be to write a simpler program which gets us part of the way to our ultimate goal. This could be a first attempt.

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

#define BUF_LEN 8
#define MAX_WORDS 6

void *emalloc(size_t size) {
   void *result = malloc(size);
   if (NULL == result) {
      fprintf(stderr, "Couldn't allocate memory!\n");
```

```
      exit(EXIT_FAILURE);
   }
   return result;
}

int main(void) {
   char* words[MAX_WORDS];
   char word[BUF_LEN];
   int i, count = 0;

   /* Note: the 7 in %7s should equal BUF_LEN-1 */
   while (count < MAX_WORDS && 1 == scanf("%7s", word)) {
      words[count] = emalloc(strlen(word)+1);
      strcpy(words[count++], word);
   }

   if (count > 0) {
      for (i = 0; i < count; i++) {
         printf("%s\n", words[i]);
      }
   }

   return EXIT_SUCCESS;
}
```

All this program does is read in a bunch of words, store them in an array, and then print them out again. The memory allocation code with associated error checking has been put into a separate function to make things clearer. By using small initial values for `MAX_WORDS` and `BUF_LEN` we can easily check that they are doing what we expect them to do when given too many words or excessively long words. Once we are happy that things are working as they should we can begin modifying our program in order to meet the specified requirements.

We will begin by adding a `print_above_average` function which first calculates the average word length and then prints out each word which is longer than the average. We could have broken this down into two steps and implemented them one at a time if we had too much time on our hands.

```
void print_above_average(char** words, int count) {
   double average = 0.0;
   int i;

   for (i = 0; i < count; i++) {
      average += strlen(words[i]);
   }
   average /= count;
   printf("Average length: %f\n", average);

   for (i = 0; i < count; i++) {
      if (strlen(words[i]) > average) {
         printf("%s\n", words[i]);
      }
   }
}
```

Once we have written this function we just need to call it from main with `words` and `count` as arguments. This should be done after the words have been read in, and only if `count` is greater than 0. We also need to change the value of `MAX_WORDS` to 100 and `BUF_LEN` to 80.

Automated testing

We could test our program by giving it its own source code to process.

```
./longwords < above-average.c
```

Which would produce output similar to this:

```
Average length: 5.500000
#include
<stdio.h>
#include
<stdlib.h>
...
```

Of course we really want to test our program with a range of different inputs and check to see if the output is correct. This can be pretty tedious, especially since it should be repeated whenever we make any alterations to our program (since we may have introduced an error).

A good way to keep our programs accurate is to make a set of input files and corresponding output files and batch test our program using a single command. This means that we can verify that our program still behaves as expected after we have made a change to it. If we find an error in our program that is produced by a certain input, we should fix our program and add that input to our test suite.

Here is an example script that you can use and modify when designing your own test setup.

```
#!/bin/bash

# This script runs a program (given as first argument) taking input from
# each file in $testdir/infiles and compares the output to the same named
# file in $testdir/outfiles.
#
# You can change the value of $testdir by giving second and third arguments
# of -d and <directory-name>.

testdir=.

if [ ! -x "$1" ]; then
   echo "Usage: $0 <program-name>"
   echo \'"$1"\' does not exist or can not be executed
   exit 1
fi

prog=$1
shift

# use alternative directory to get testfiles from
```

```
if [ "$2" == "-d" -a -d "$3" ]; then
   testdir=$3
   shift 2
fi

indir=${testdir}/infiles
outdir=${testdir}/outfiles
tmpfile=temporary-resultfile

declare -i numtests=$(ls ${indir} 2>/dev/null | wc -w)
echo "Running black box tests"
echo "-----------------------"
echo "Test directory = $indir"
echo "Number of testfiles = $numtests"

if [ "$numtests" -gt "0" ]; then
   echo
fi

for file in $(ls ${indir} 2>/dev/null)
do
   if [ ${file%\~} != ${file} ]; then # ignore ~ files
      continue
   fi
      if [ ! -f ${outdir}/${file} ]; then
         echo Error: no match for \'${file}\' found in \
            directory "'"${outdir##*/}"'"
         continue
      fi
      ./$prog $@ < ${indir}/${file} >| ${tmpfile}
      diff ${tmpfile} ${outdir}/${file} > /dev/null
      if [ "$?" -ne "0" ]; then
         echo ${file} failed!
         echo
         echo Showing difference for ${file}
         echo "< your output"
         echo "> expected output"
         echo
         diff ${tmpfile} ${outdir}/${file}
         echo
      else
         echo ${file} passed.
      fi
done
```

This script is written in BASH which is the default shell under Linux and OS X. The syntax can take bit to get used to but it's worth making the effort to do so because shell scripting is a very useful tool. Whenever you type something into a terminal you are using BASH. There are many resources on-line which provide a huge range of information about programming in BASH. In addition to this, as always, the laboratory teaching staff are happy to help you with any questions you have.

Checking for memory errors

There are a couple of errors which are particularly easy to make when programming in C. The first is not deallocating all of the memory that your program allocates (this is called a memory leak). The second is accessing areas of memory outside those which you have allocated (e.g. overflowing a buffer). If you are using Linux there is a program called *valgrind* which can find memory errors in your programs. To use it just put `valgrind` before the name of your program and run it as you normally would.

```
valgrind ./longwords < above-average.c
```

This produces the same output as when we run it normally, but valgrind checks what happens with memory allocation and deallocation and prints out a summary at the end.

```
==13760== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 11 from 1)
==13760== malloc/free: in use at exit: 650 bytes in 100 blocks.
==13760== malloc/free: 100 allocs, 0 frees, 650 bytes allocated.
==13760== For counts of detected errors, rerun with: -v
==13760== searching for pointers to 100 not-freed blocks.
==13760== checked 56,316 bytes.
==13760==
==13760== LEAK SUMMARY:
==13760==   definitely lost: 650 bytes in 100 blocks.
==13760==     possibly lost: 0 bytes in 0 blocks.
==13760==   still reachable: 0 bytes in 0 blocks.
==13760==        suppressed: 0 bytes in 0 blocks.
==13760== Rerun with --leak-check=full to see details of leaked memory.
```

Whoops! We forgot to free up the memory that we allocated before our program finished. All we need is a simple loop before our final return statement

```
while (count > 0) {
   free(words[--count]);
}
```

and all is well again.

```
==13965== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 11 from 1)
==13965== malloc/free: in use at exit: 0 bytes in 0 blocks.
==13965== malloc/free: 100 allocs, 100 frees, 650 bytes allocated.
==13965== For counts of detected errors, rerun with: -v
==13965== All heap blocks were freed -- no leaks are possible.
```

In summary, we have looked at a number of aspects of writing programs. Before anything else, we must define what our program is to do. Next, we can build up our program piece by piece to make sure that it works correctly. And, as part of our testing, we should automate our testing process so that we can easily verify at any stage if our program is working correctly. Finally, there are various tools we can use to further check our programs for correctness.

Appendix B

More about pointers

If you have done Cosc 243 you will remember that bytes are stored in groups, called words, in main memory. However, our program can still access each individual byte. Currently there are either 4 or 8 bytes to a word on most desktop computers. In figure B.1 below you can see our main memory arranged in words of 4 bytes in length, hence the addresses our program can access increments by 4 bytes to each subsequent word. Now remember that an int is 32 bits and so when we *declare* an int, such as with `int i`, our program finds a whole word in main memory to set aside for it. In figure B.1 we also *initialised* that int to hold the value 5. We also declared a double (`double d;`) which set aside 2 words (remember that a double is 64 bits = 8 bytes long) for that variable. Let us say that on our system, an address is 4 bytes long, so that when we initialised our int pointer (which is to hold an address of an int) we got allocated a single word. Once we've declared any of these variables we can print out their values like this:

```
printf("i=%d, p=%p, d=%f\n", i, p, d);
```

[Note that the placeholder for our pointer is `%p`. Also note that if you try this you will get compilation warnings: you can ignore these for now as long as you don't have any errors. We will discuss these warnings more below.]
At the moment, however, only the variable i will hold anything sensible. There will be values for both p and d but they will be just whatever happened to be in those memory locations at the time they were allocated them. We can also print out the memory address of each of the variables that we've declared:

```
printf("address of i=%p, p=%p, and d=%p\n", &i, &p, &d);
```

Next in figure B.1 we give the double variable a value of 5.0. We also assign the address of our int variable i to the int pointer variable p. So the int pointer variable p now holds the *address* of the int variable i. We say that "p points to i". We could now print out the value of p and see that it has changed to be the address/location of i.

Lastly in figure B.1 we try to assign the address of the double variable d to the int pointer variable p. However, as we declared p to be an int pointer, our compiler is expecting that we only assign addresses of ints to it, not addresses of doubles. Thus we would get a compilation warning. The program will still compile and run with p now holding the address of variable d.

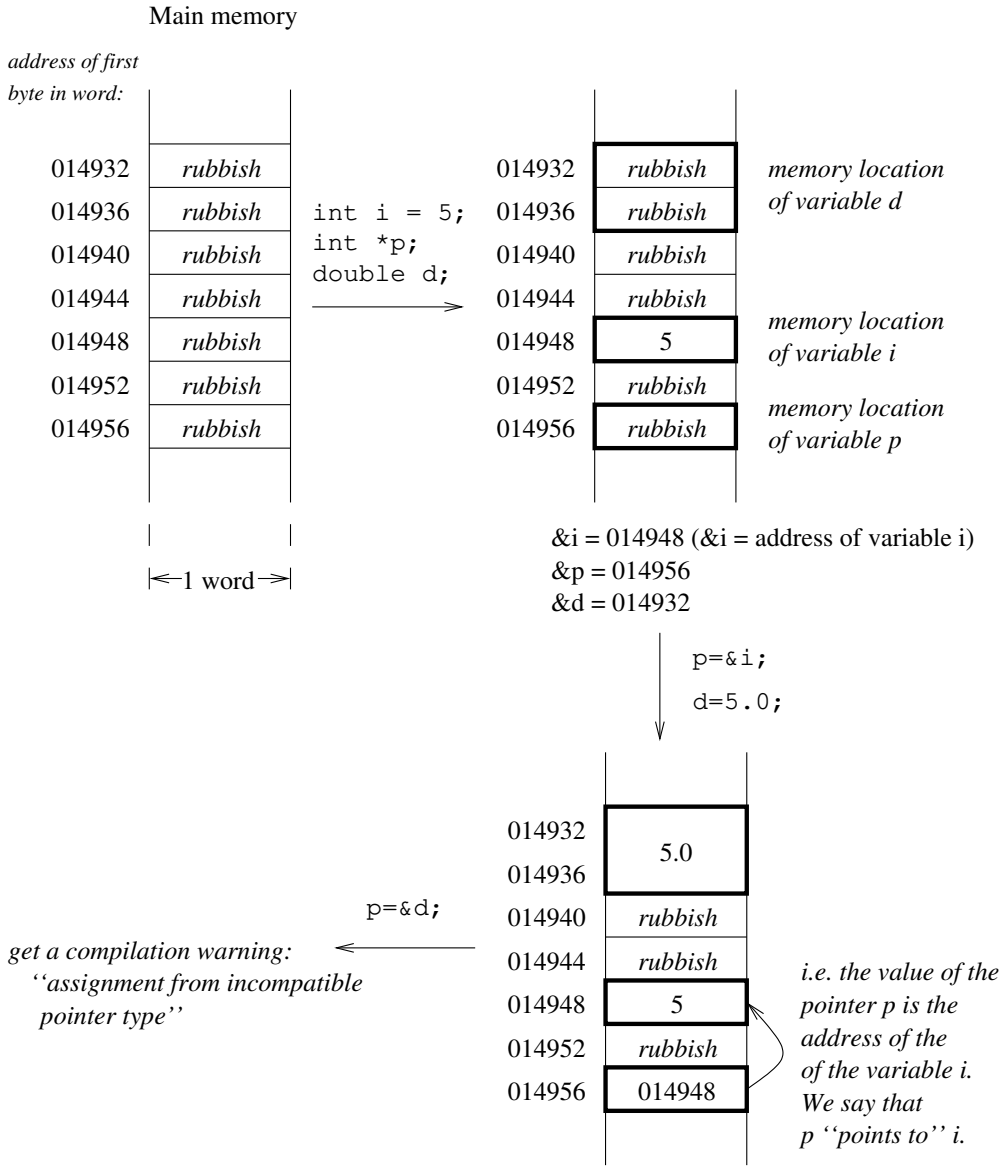


Figure B.1: A pointer to an int can only point to an int.

Pointer types and void pointers

Okay, so we know that we get a warning if we try to make an int pointer (such as p declared in figure B.1 above) point to a double. If we wanted to create a pointer that could point to our double variable we could have declared it like this:

```
double *dp;
```

But what if we want a general pointer that won't be so fussy and will simply hold an address, and not worry about what is at that address? Let us introduce void pointers. Void pointers point to a bit of memory and don't care what is at that address. We could declare one like this:

```
void *vp;
```

If we had used this void pointer variable vp instead of the int pointer variable p in the figure B.1 example above, we would not have got any warnings.

Now, how about all those warnings that we got when we tried to print out our pointer, and the other addresses? Well, this is because in the printf function, %p is expecting a void pointer, not a pointer to an int, or addresses of doubles or even of addresses of pointers themselves. But, of course, all these things are just addresses and we can simply cast their values to void * (void pointers) if we wish:

```
printf("i=%d, p=%p, d=%f\n", i, (void *) p, d);
printf("address of i=%p, p=%p, and d=%p\n",
         (void *) &i, (void *) &p, (void *) &d);
```

Converting between pointers, variables and addresses

In figure B.1 above, we put in p the address of i. Now let's say we want to get the value stored at the address in p, i.e. we want our program to first go and look up the value of p and then follow that address and give us the value located at that address. To do this we *dereference* the pointer p like this:

```
*p
```

So we could print out something useful like this:

```
printf("Value located at address %p is %d\n", (void *) p, *p);
```

Figure B.2 shows a diagrammatic way of remembering how to do these conversions. Remember you have already encountered the use of & to get an address.

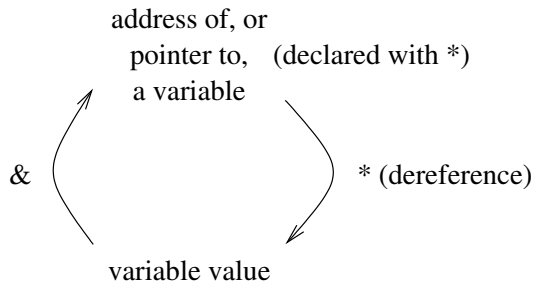


Figure B.2: Converting between addresses and variable values.

Arrays

When we declare an array like this:

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

our program finds a bit of contiguous memory 4 bytes/int $\times$ 5 ints = 20 bytes long (see figure B.3 below). The variable a is considered a *permanent* pointer to the first byte in this allocated memory. Thus a is an address (014940 in our case).

If we don't know the size of our arrays until run-time then we have to allocate them the correct amount of memory manually:

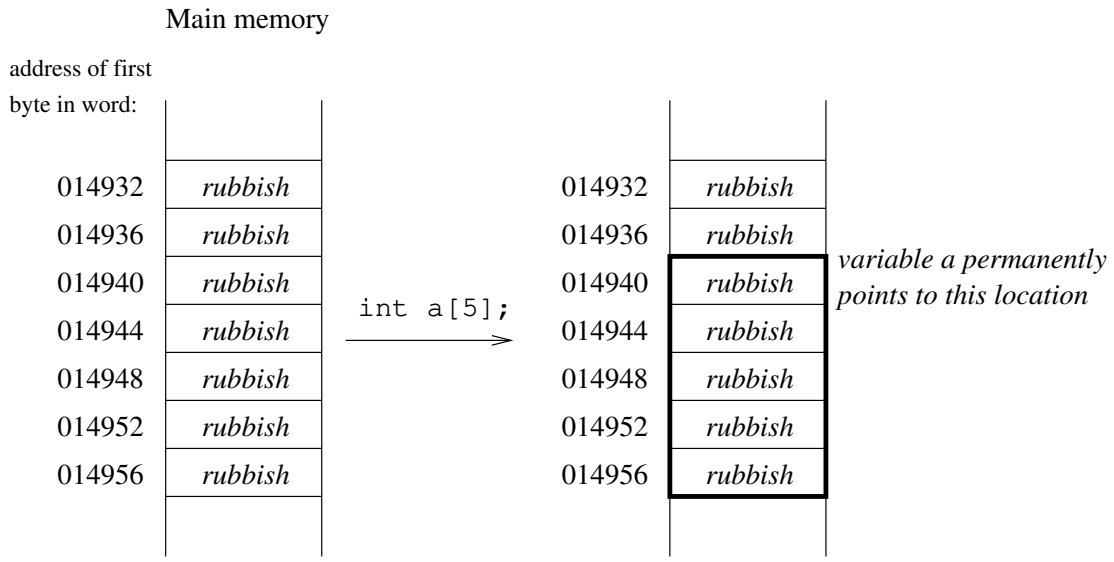


Figure B.3: Declaring an array.

```
int *a;
a = malloc(5 * sizeof a[0]);
```

When we declare `a` as `int *` we find a bit of memory to hold an address (see figure B.4 below).

Next when we malloc some space we find a bit of contiguous memory `5 * sizeof a[0]` bytes long (our program knows that there should be an int at a[0] as a is an int pointer, so will return to us the correct number of bytes: we use `sizeof a[0]` rather than `sizeof (int)` as we may wish to change a to a double * or char * at some later point and we don't want to have to change too much else in our code).

The main difference with this dynamically created array is that we can change what our pointer, a, is pointing to: it is not permanent like in figure B.3 above.

Now to get the values stored in either of these arrays we can either use them directly like an array, e.g.

`a[0], a[1], a[2], a[3], a[4]`

will give use the values stored in each position in the array (of course we'd have to put something sensible in there first to be of any use). Or, we could access them by dereferencing their pointers. So `a` is the pointer to (address of) the first variable, `a+1` will be the pointer to (address of) the next variable, and so on. So, an equivalent way to get the values would be:

```
*a, *(a+1), *(a+2), *(a+3), *(a+4)
```

Arrays of pointers versus multi-dimensional arrays

Refer to sections 5.6 to 5.9 in Kernighan and Ritchie "The C programming language".

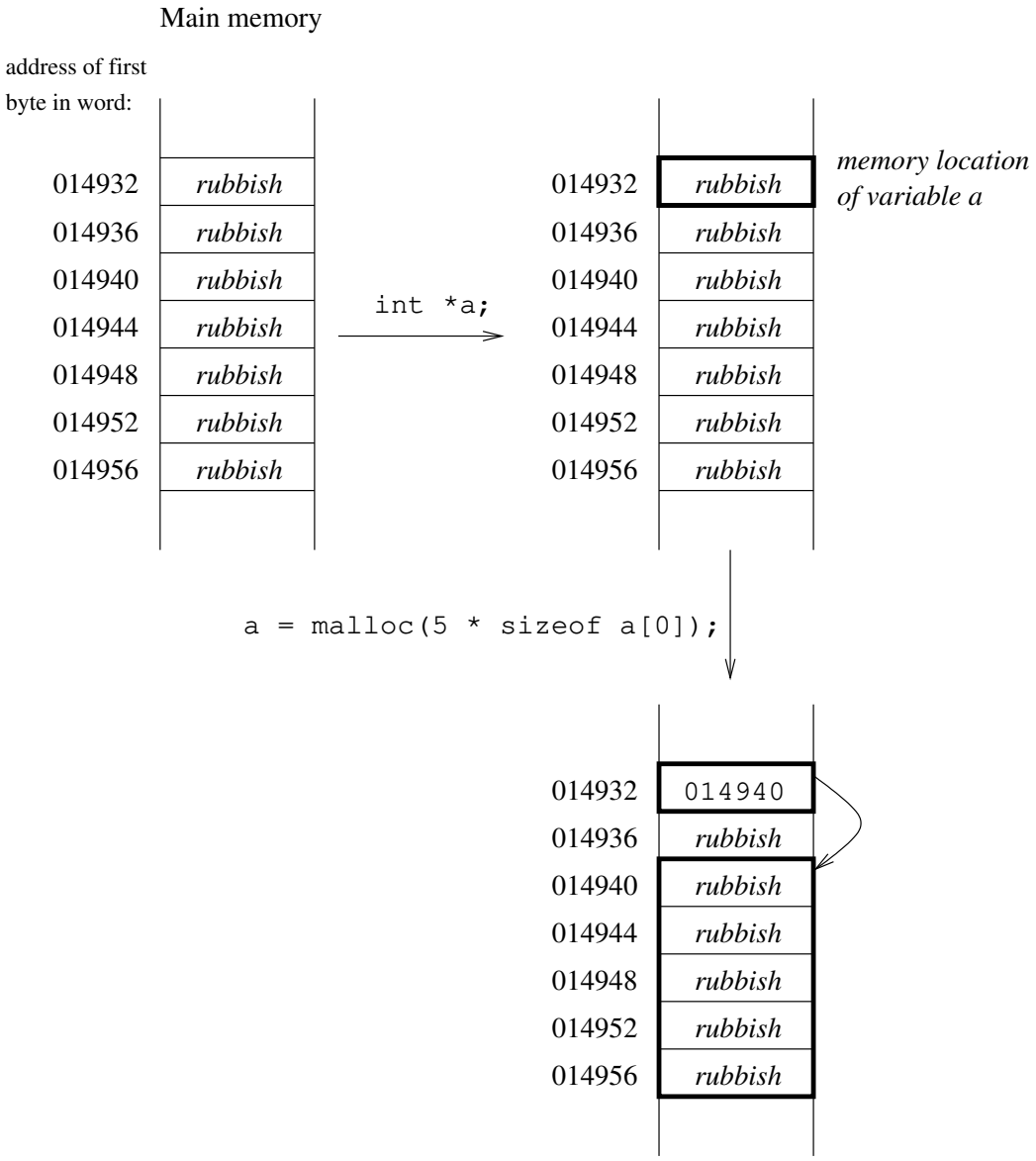


Figure B.4: Dynamically creating an array.

Appendix C

Debugging C programs with GDB (GNU debugger)

(Note: that the output of gdb will vary slightly depending on the platform it is run on.)

It's a fact of life that everybody makes mistakes while writing code. Sometimes we know there's a bug, but by just reading the code we can't seem to discover it. We can try using print commands to locate the error, or we can launch a debugger when we get into trouble. It shouldn't replace making an effort to write correct code, or adding tests in the code for invalid function arguments, NULL pointers, etc. But there are times when it is the best option.

The GNU debugger, gdb, is very useful when one of your programs continually seg faults and you don't know why. With gdb you can run your program inside a controlled environment – you can arbitrarily start and stop execution, watch the values of variables change, trace execution one line at a time, and even analyse the results of a program crash to see what went wrong.

This appendix walks through a simple gdb debugging session. The code being debugged is very basic, but the same commands and principles apply to much more complicated programs. Start by copying the file `$C242/gdb/buggy.c` (listed below) into one of your own directories.

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

void foo(int *bar) {
   *bar = 3;
   printf("bar is %d\n", *bar);
}

int main(void) {
   int *baz;

   baz = malloc(sizeof *baz);
   if (baz = NULL) {
      return EXIT_FAILURE;
   }
```

```
    foo(baz);
    return EXIT_SUCCESS;
}
```

Now let's compile it and see what happens.

```
$ gcc -W -Wall -O2 -ansi -pedantic -o myprog buggy.c
buggy.c: In function 'main':
buggy.c:13: warning: suggest parentheses around assignment used
    as truth value
```

Okay, we have a warning that recommends adding some parentheses. It's a bad idea, but let's do that to the if statement:

```
if ((baz = NULL)) {
```

Now if we compile our file again we don't get the warning anymore. So let's run it.

```
$ ./myprog
Segmentation fault
```

Whoops! Something in the program is definitely wrong. Before we can start up gdb, we have to recompile the program with debugging symbols enabled. The -g option tells most compilers to include symbolic information about the program in the executable file. This makes it easier for the programmer to figure out what's going on since a value makes more sense than 0x804a088.

```
$ gcc -g -W -Wall -O2 -ansi -pedantic -o myprog buggy.c
```

Now that we have a version of our program suitable for debugging, we can open it inside gdb.

```
$ gdb myprog
GNU gdb (GDB) Fedora 7.12-36.fc25
Copyright (C) 2017 Free Software Foundation, Inc.
...
```

The (gdb) prompt tells us that gdb is ready and awaiting commands. Typing *help* at the prompt will give you a list of available commands. The first thing we want to do is run the program inside of the debugger to see exactly where it dies. (If the program requires command line parameters, we can supply them to the *run* command of gdb. For example: `run 100`, or redirecting input from a file `run < data.txt`)

```
(gdb) run
Starting program: /home/cshome/a/astudent/242/myprog

Program received signal SIGSEGV, Segmentation fault.
0x080483a9 in foo (bar=0x0) at buggy.c:5
5         *bar = 3;
```

According to gdb, the error is occurring on line 5. This line is inside the `foo()` function and is a dereference of the pointer bar to set its contents to 3. Our next step is to use the *list* command to print the code surrounding line 5 so we can get an idea of where we are in the program.

```
(gdb) list
1       #include <stdio.h>
2       #include <stdlib.h>
3
4       void foo(int *bar) {
5         *bar = 3;
6         printf("bar is %d\n", *bar);
7       }
8
9       int main(void) {
10        int *baz;
```

Setting a breakpoint in the `foo()` function will halt the program's execution and let us step through the code one line at a time. A breakpoint can be set either by specifying the name of the function or a line number - the example below uses the name of the function, but the command *break 5* would do the same thing.

```
(gdb) break foo
Breakpoint 1 at 0x80483a6: file buggy.c, line 5.
```

Now if we run the program again, it will stop at the beginning of `foo()` and wait for further instructions.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/cshome/a/astudent/242/myprog

Breakpoint 1, foo (bar=0x0) at buggy.c:5
5         *bar = 3;
```

Sometimes using the *print* command to examine the values of variables can provide a clue as to what went wrong. Let's look at the bar variable used in the `foo()` function.

```
(gdb) print bar
$1 = (int *) 0x0
```

The value of bar is not what we would expect. The `(int *)` means that bar is a pointer to an integer, but `0x0` tells us that this pointer is pointing to memory location zero, or `NULL`. The `foo()` function is so simple that we're pretty sure it's not the problem, so something in `main()` must be causing our error. Let's use a breakpoint on line 13 to check what happens after we allocate memory using `malloc()`.

```
(gdb) break 13
Breakpoint 2 at 0x80483f2: file buggy.c, line 13.
```

Running the program again, will stop at the beginning of the if statement and wait for further instructions.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/cshome/a/astudent/242/myprog

Breakpoint 2, main () at buggy.c:13
13        if ((baz = NULL)){
```

We should check the pointer baz to see if `malloc()` has allocated any memory.

```
(gdb) print baz
$2 = (int *) 0x8c5c008
```

It looks like that `malloc()` has done its job correctly. If we run through the code one line at a time using the *next* command we should be able to see what happens to baz.

```
(gdb) next
16        foo(baz);
```

We get to the point that `foo()` is going to be called, with a reference to the baz pointer. Let's check it again.

```
(gdb) print baz
$3 = (int *) 0x0
```

Hmmm, looks like the memory allocated to baz has disappeared. We can fix this using the *set* command to see if this is the only problem in our code.

```
(gdb) set baz = malloc(sizeof baz[0])
```

Just to be sure we have some memory for our pointer.

```
(gdb) print baz
$4 = (int *) 0x8c5c018
```

Good, so we have memory for our pointer. To delete a breakpoint we use the *clear* command with a function name or a line number (to delete all the breakpoints use the *delete* command).

```
(gdb) clear foo
Deleted breakpoint 1
```

It is time to *step* into `foo()` and see if there are any more problems.

```
(gdb) step
foo (bar=0x98e0018) at buggy.c:5
5         *bar = 3;
```

Let's check the bar parameter again.

```
(gdb) print bar
$2 = (int *) 0x98e0018
```

Looks good as bar is not 0X0 anymore. To run the rest of program use the *continue* command.

```
(gdb) continue
Continuing.
bar is 3
Program exited normally.
```

Perfect, our program has finished without any problems. Did you spot what the problem was? What we have in the condition of the if statement is setting the baz pointer to zero: `if (baz = NULL)`. I am sure you realised that an equals sign for the comparing baz with NULL is missing, so it should be:

```
if (baz == NULL)
```

What was happening is that NULL has a zero value in C, so `baz = NULL` sets baz to zero. As we know zero is interpreted as false in the if statement, so what is in the if statement block never was executed.

Finally to exit gdb type *quit* to return back to command line prompt.

```
(gdb) quit
```

For more information about gdb, use the man pages, by typing `man gdb` in a terminal, or use gdb's own help system. There is also a file which contains a good summary of many gdb commands in the `$C242/gdb` directory.

Appendix D

Assessment details

Assessment	Due Date	Description	Weight
Lab 4	Jul 17^{th}	*Sorting Arrays*	2%
Lab 7	Jul 28^{th}	*Practical Test 1*	3%
Lab 13	Aug 18^{th}	*Practical Test 2*	8%
Assignment	Sept 14^{th}	Programming assignment	15%
Lab 22	Sept 25^{th} or Sept 29^{th}	*Practical Test 3*	12%

There will be a chance to resit one or more of the practical tests on Tuesday Oct 6^{th}