IOI Training 2017 - Week 0

Payton Yao

February 27, 2017

1 Introduction

Throughout this first week, we'll be delving into what happens when you run programs. We'll have a look at what happens when you compile your program, what it looks like when you run them, and reasons why that's important. We'll have some experiments and exercises to understand what is really happening.

2 Command Line

Many of you already know how to code. Some way or another, you've managed to solve some problems from the NOI allowing you to get this far. For some of you, you might have programmed directly into HackerRank's web interface, compiling and debugging your programs through that platform. For others, you might have used an Integrated Development Environment (IDE) like CodeBlocks or DevC++. For those unaware, IDEs combine multiple tools in one, allowing you to edit source code, compile programs, debug them, and more, all in one convenient program. From our understanding, very few of you do everything through the command line which is what we'll introduce here.

You might be asking: "Why?" If everything can be done on an IDE, why bother learning to use the command line? There are two answers, both equally valid. The first and more practical one is that you may not have the exact IDE you're comfortable with available at all times. Especially at the IOI, HackerRank definitely won't be available. Every computer will have some form of command line available, and every computer meant for programmers will have the command line tools set up. The second reason is about understanding. IDEs give way to too much magic and sometimes we miss all the small things that happen behind the scenes.

Note that this is only a short introduction into the command line. Mastering the use of the command line takes years of experience to do extremely intricate things, but basic usage allowing you to navigate the file system, compile code, and run programs you can learn in the space of an hour or so.

At this point, it's important to note that there are some major differences between using the command line on Windows and on Linux. For Mac users, you can follow along on the Linux path since the differences are small. Please read the appropriate section based on your operating system.

As a historical note, there was once an operating system called Unix developed at Bell Labs in the 1970s. It's this original operating system whose design philosophy Linux and Mac copied. Because they come from similar roots, much of the core command line tools are the same with some small differences in some of the options they can be given. Advanced users will also note a striking similarity in the way system files are organized, for example Linux has `/home/` where Mac has `/Users/`.

2.1 Windows

On Windows, you have two main choices of terminals: Command Prompt (`cmd.exe`) and PowerShell (`PowerShell.exe`). PowerShell is the newer (and more powerful) one. Command Prompt has a width limit of 80 characters, so it's kind of annoying. Run them from the Start Menu or the Windows 8 search.

The important commands are as follows:

- `cd` - change directory, use as `cd .`

- `dir` - list files in directory, use as `dir C:\Users\Public`
- `copy` - copy a file, use as `copy a.txt b.txt`
- `move` - move a file from one place to another, use as move original.txt new.txt. Interestingly, this is also the command used to rename files.
- `del` - delete a file, use as `del a.txt`
- `mkdir` - make a new directory, use as`mkdir new_directory`

We encourage you to try these commands out. Your computer won't break if you type a command wrong. You can hit the tab key to auto-complete.

Alternatively, you can try out the new Linux subsystem. You can find out how to install it here: https://msdn.microsoft.com/en-us/commandline/wsl/install_guide

2.2 Linux and Mac

For Mac users, you can use the built-in Terminal, just search for it using Spotlight. You can also opt to download something called iTerm2 which is a much improved version of Terminal with lots of useful new features. For Linux users, you'll most likely be using the Gnome Terminal if you're on Ubuntu. (And if you're not on Ubuntu, you probably already know what you're doing anyway.)

The important commands are as follows:

cd - change cirectory, use as `cd /Downloads`. ls - list files in current directory, use as `ls` cp - copy a file, use as `cp a.txt b.txt` mv - move a file from one place to another, use as `mv original.txt new.txt`. Interestingly, this is also the command used to rename files. rm - delete a file, use as `rm a.txt` mkdir - make a new directory, use as `mkdir new_directory` rmdir - delete an empty directory, use as `rmdir new_directory` Again, we encourage you to try these commands out. Your computer won't break if you type a command wrong. You can hit the tab key to auto-complete.

You may have noticed that we did `cd /Downloads`. `/` is the Linux shorthand for `/home/user/` and is the Mac shorthand for `/Users/user/`. We'll have a bigger discussion on Unix commands and the environment in the future, but this is enough for now.

Read more about the unix command line at:
http://www.ee.surrey.ac.uk/Teaching/Unix/.

3 Compiling and Executing Programs

For this training, we will focus on using C++. And to run C++ programs, we first have to learn to compile them. And to compile them, you first need to install a compiler! It's different on every operating system, so we'll just link you to instructions written by others.

3.1 Installing

3.1.1 Windows

Please go to https://wiki.wxwidgets.org/Installing_MinGW_under_Windows. There are more detailed instructions on the link. Make sure you don't forget to add `C:\MinGW\bin` to your Path environment variable.

3.1.2 Mac

For Mac users, you have to install Xcode then download the command line tools from there.

3.1.3 Linux

For Linux users, g++ comes installed on the base operating system.

3.2 Using g++

Now that you're all done setting up, it's time to start compiling! If you save your source code file as `source.cpp`, you can compile it using the command `g++ source.cpp`. By default it saves the resulting executable as `a.out`.

On Windows, the default name of the executable is `a.exe`. You can change the name for the resulting output file by adding `-o filename` to the command. For example, if my source code is `example.cpp` and I wanted to save the output executable as `executable` I should then type the command `g++ example.cpp -o executable`. Note that for Windows users, the resulting executable name should end with `.exe` otherwise Windows won't run it.

We recommend that you try compiling a sample Hello World program this way before moving on.

If you're having an error like `g++: error: source.cpp: No such file or directory`, please ensure you're in the right directory first.

3.3 Running Executable Files

Now that you've compiled your program, it's time to run it! For Linux and OSX users, the command to run a program is `./program_name`. So if your program is saved as `a.out`, you should type `./a.out`, and if it's saved as executable, you should type `./executable`. For Windows users, you don't even need the `./`. So if your program is saved as `a.exe`, just type `a.exe` directly and it should run.

3.4 Killing Running Programs

In case you accidentally run code that never ends or you just want it to die for some other reason, what can you do? It's gonna block your command line and stop you from typing other commands, so you will want to kill it. You can opt to kill it using the task manager, but there's a much easier way. Just press Control + C. This will work on any operating system and will kill the program that's hogging up the command line.

Please write a program with an infinite loop, compile it, run it, then kill it just to try out Control + C.

4 Input redirection

4.1 Saving Output to a File

There are cases where we want to save the output of our program to a file. The simple way is to highlight and copy things from the command line, paste it in notepad, and save the file. But there's an easier and more precise way to do it. Use the `>` symbol.

To be precise, if we want to save the output of the program executable to a file called output.txt, we type the command `./executable > output.txt`.

On windows, the corresponding command is `executable.exe > output.txt`.

Try it yourself! Of course, you can change the output file name to anything you want and not just `output.txt`

It's important to note at this point that **this will completely erase the contents of the file you're saving into**, even if your program hasn't printed anything out yet. So be careful!

4.2 Reading Input from a File

Another common case is when we have input from a file that we want our program to read in from instead of us having to type the values. Again, there's a simple way to do that! Use the `<` symbol.

If we want to make the program executable read from a file called input.txt, we type the command `./executable < input.txt`.

On windows, the command is `executable.exe < input.txt`.

This is extremely useful for debugging your solutions for contests. Especially for problems with a lot of input like graphs.

4.3 Combining the Two

There are cases where we have input from a file and we want to save the output to another file. During the Google Code Jam, input files are downloaded from their website and you have to upload the correct output file. Or maybe you just want to store the output to a pre-determined input. The way you do that is just combining the two.

An example command that combines them is `./executable < input.txt > output.txt`

Note that the command `./executable > output.txt < input.txt` is just as valid and does exactly the same thing.

I'm sure the Windows users among you can get the pattern by now, so I won't belabor the point.

4.4 Extras: pipe

On the topic of redirection, sometimes you will find that you have two programs and you want the output of one file to feed into the input of the other. The simple solution is to redirect the output to a file then run the second one with the output of the first. You can save yourself the temporary file by using the `|` symbol. For example: `./program1 | ./program2`

4.5 Extras: diff

There's a very useful tool available on Linux and Mac called `diff`. What `diff` does is it compares two files line by line and tells you which lines differ. The usage is `diff file1.txt file2.txt`.

It's useful to consider using if you have, for example, a slow solution and a faster but more complex one. You can use the slow solution for a lot of small cases then use diff to ensure that the slow solution and the faster one give the same results.

5 Random Access Memory

Modern computers have something called Random Access Memory (RAM). Most modern computers have anywhere between one gigabyte to thirty two gigabytes of this, and it's the topic of our discussion today. At its core, random access memory is hardware that stores data. We won't go into details of how it works on a physical level, but at an abstract level it does two things. It responds to a command saying "Store these 8 bits in slot X" and "Retrieve the 8 bits in slot X". The number of slots correspond to how many gigabytes the RAM is.

The important thing to note is that "Store these 8 bits in slot X" makes no difference whether your 8 bits are supposed to represent integers or floats or strings. RAM simply does not care. It takes those bits and shoves them into the buckets. If you wanted to store more than 8 bits, it's going to take more instructions.

Okay, this is an oversimplification, there are actually 8 instructions, two of which are "Store these 8 bits in slot X" and "Store these 32 bits in slots X, X+1, X+2, X+3" and the rest follow that simple pattern, with the RAM supporting 8-bit, 16-bit, 32-bit, and 64-bit operations.

It's interesting to note that, RAM is the reason why "try turning it on and off again" works. If RAM stored water in buckets, you can think of the buckets as having holes in them. If the power is turned on, someone is always refilling the filled buckets. But when the power goes off, all the buckets drain to empty, and whatever messed up state your computer was in is completely wiped out. And RAM is manufactured this way because of economic and technological reasons.

Again, we emphasize **all memory is the same**. There is no separate memory for integers or floats or strings or arrays. The only difference is how we interpret regions of memory. Eight bytes can be taken to mean a string of eight `char`s, a single `long long`, two `int`s, or one `double`. What matters is what your program thinks it is.

6 Pointers

Pointers allow you to access the slot numbers themselves. The official term used is the **address**. Every variable has an address which is a 64-bit number. Some older computers from the early 2000s might still

have addresses as 32-bit numbers because RAM rarely exceeded four gigabytes back then, but for the most part addresses nowadays are 64 bits long.

The address of a variable can be gotten by using the `&` operator. One naive way to store it is as a `long long`, but as we mentioned earlier, does this number refer to a `char` or a `long long` or maybe something even bigger?

In C++, the pointer type is what is used to tell the computer how we want to interpret these 64-bit addresses. To declare a pointer variable, we declare a variable with a `*`. For example, we can do:

```
void main() {
  int x;
  int* pointer_to_x = &x;
}
```

In case any of you become confused with examples on the internet, most other sources will put the `*` declaration beside the variable name as in `int *pointer_to_x`. Both of these mean the same thing, but I find more meaning in the way I prefer to do it. Unfortunately, the designers of the original C prefer the second because `int *ptr, x` makes a pointer to an int and then a normal int. If you wanted two pointers, you'd have to declare `int *ptr, *x`. I will not be doing this in any of our examples because I believe that the variable type is a "pointer to an int" and not an "int that happens to be a pointer". This is a stylistic preference, so do what you want. The C++ compiler doesn't care how you put the spaces.

Of course, you can do this even more.

```
void main() {
  int x;
  int* pointer_to_x = &x;
  int** pointer_to_pointer_to_x = &pointer_to_x;
  int*** pointer_to_pointer_to_pointer_to_x = &pointer_to_pointer_to_x;
}
```

At some point it gets a bit silly. But we will find a use for this kind of thing later when we talk about arrays.

To access the memory at that address, we can use the `*` operator, known as the **dereference operator**. Don't get confused by the multiplication operator with the same symbol, the dereference operator is unary, similar to how the unary negative operator is different from the minus operator.

The type of the pointer tells the computer how many bytes to interpret starting from that address. For example, look at the following:

```
#include <iostream>
using namespace std;

int main() {
  unsigned int x = (3 << 24) + (5 << 16) + (7 << 8) + 11;

  char* ptr = (char*)&x;

  cout << (int)*ptr << " ";
  ptr++;
  cout << (int)*ptr << " ";
  ptr++;
  cout << (int)*ptr << " ";
  ptr++;
  cout << (int)*ptr << " ";
}
```

This code prints out "11 6 5 3 " because of something called "endianness". But the important point is that we can abuse different pointer types to access the same data in different ways. **All memory is the same.**

Now you may be wondering why we would even want to mess with the memory address of variables instead of the variables directly. The above example is clearly really contrived. So let's give a simple one for the sake of example. Suppose we wanted a function to return two variables. But C++ only allows one variable at a time. The more clever ones among you might consider packing multiple values in a single 64-bit number. The ones who know a bit more C++ might think of using structs. But there's something done in the standard C library that uses pointers, and I think it's pretty great. The function in question is called `scanf`. You use it like: `scanf("%d %d", &a, &b")`. Note that it asks for the address of the variables it should store the data in. The implementation of such a function would look like this:

```
#include <iostream>
using namespace std;

void foo(int* addr1, int* addr2) {
  *addr1 = 5;
  *addr2 = 7;
}

int main() {
  int a, b;
  foo(&a, &b);
  cout << a << " " << b;
}
```

7 Arrays and Pointers

7.1 Arrays as Blocks of Memory

An array is a big continuous chunk of memory. To get the first item in an array of 1-byte objects, we just look at the first byte of that chunk of memory. The second item is the second byte. And in general the nth item is the nth byte in that chunk of memory.

If we have an array of 4-byte objects like `int`s, then the first item is the first four bytes of memory. The second item is bytes five to eight. And so on.

Another way to view it is if we have a pointer to the first byte of memory in that array, then we can find a simple formula that makes the pointer point to the nth byte of that array. If we have 1-byte objects, we only need to look at ptr + (n-1).

7.2 Pointer Arithmetic

Extending this idea, you would think that if we have 4-byte objects, to get the nth item in the array, we want to look at bytes ptr + 4*(n-1) until ptr + 4*(n-1) + 3. But it's not!

If you have a pointer and you add an integer to it, it automatically multiplies the added integer by the size of type of the pointer. So an int pointer, if you add 1 to it, will actually get incremented by 4. If you add 3 to it, will get incremented by 12. In some way, this makes pointers easier to work with if you think about them as arrays. Some people might find this strange, but that's just the way C++ was designed. This is called pointer arithmetic, by the way. Most of the time, you just want to increment and decrement pointers, and it generally works out really well. In case you really need to mess with memory addresses (which you almost never should), you can just typecast your pointers into long long before doing the operations, then typecast them back to your prefered pointer type.

To repeat that previous paragraph, if we have a variable `int* p` and we know `p == 100`, `p+1 == 1004`, `p+3 == 12` because of the quirks of pointer arithmetic. Explicitly turn them into `long long` if you want to do some black magic with your memory addresses.

7.3 Zero-based Indexing

Earlier we mentioned that to get the nth item in an array, we look at ptr + (n-1). But if we just started counting from 0, then we can save a subtraction! The 0th item is at address ptr, the 1st item is at address ptr + 1, the 2nd item is at address ptr + 2. And the kth item is at address ptr + k. This trick is the reason why programmers like to count from 0.

As a side note, there are some programming languages that are not zero-indexed and their compilers automatically add -1 when arrays are accessed. That's not C++ though, so no need to worry about it.

7.4 The [] Operator and the * Operator

By now you might be thinking: "Wait a minute! I've been doing array access without all this pointer stuff. It's just arr[x] right?" And I'm here to tell you that array access by doing `arr[x]` is completely the same as `*(arr+x)`!! The compiler turns them into the exact same processor instructions. So if you have a pointer that points to a single variable and not an array, instead of doing *ptr you can just as well do `ptr[0]`. It might not make intuitive sense when you're reading some code that's not yours though. Most people will expect `[]` for arrays and `*` for single variable pointers.

8 Dynamic Memory Allocation

8.1 Stack Memory

So far we've been playing around with using pointers to point to individual bytes of an int or to access indices of an array. We also discussed how pointers can be used to implement a function like scanf. Let's look at another application of pointers: dynamic memory allocation. Normally, C++ code will only be able to access a few megabytes of memory depending on the operating system and the specs of the computer. This memory, called the **stack**, is used for all local and global variables. We can verify this by testing the following code:

```
#include <stdio.h>

int total_allocated = 0;
void go_deeper() {
  char arr[10000];
  total_allocated += 10000;
  printf("I have allocated about %d bytes of memory.\n", total_allocated);
  go_deeper();
}

int main() {
  go_deeper();
}
```

On my computer, this code goes deep enough to allocate about 8 megabytes before it crashes. But my computer has several gigabytes of memory. So what happened? And how do we access the rest of our several gigabytes of memory (or hundreds of megabytes in contests)?

8.2 The malloc Function

Let's first discuss the question of how to access the rest of our memory. There's a function called `malloc` that asks the operating system for some number of bytes, and the function will return a pointer to the start of that chunk of memory. It's up to you to decide what you want that memory region to mean. The operating system only cares that you asked for N bytes so it'll give you N bytes. If you point an int pointer to it, your program will think of it as an array of integers. If you turn it into a char pointer, it'll think it's an array of characters. Let's have a look at some sample code:

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

int total_allocated = 0;
void go_deeper() {
  char* arr = (char*)malloc(10000);
  // This malloc call gives you 10000 bytes to play with.
  // It doesn't have any idea of the meaning of this chunk of memory,
  // so it's your job to convert it to a pointer of the correct type.
  total_allocated += 10000;
  printf("I have allocated %d bytes of memory.\n", total_allocated);
  go_deeper();
}

int main() {
  go_deeper();
}
```

If you open Task Manager while this program is running, you'll notice that it's slowly eating up more and more of your memory. You might want to kill it before it goes too far and your computer starts lagging. Or you could also wait for your computer to crash or the program to crash because of memory issues. :)

8.3 The free Function

Now that we know how to allocate memory, we have to know how to de-allocate it. This is done through the free function. Just call it and pass the memory address given by malloc and it'll return the memory back to the operating system.

8.4 Heap Memory

Memory allocated and freed through the use of malloc and free is memory from the **heap**. In contrast to stack memory, heap memory is shared by all running programs. If we have time in the future, we can go into how stack and heap memory are used by the compiler, but it's not really necessary. The important thing to know is that **Stack memory is limited to a few megabytes. Heap memory is practically all the memory available on your computer. Stack memory is used for local variables. Heap memory is only from memory gotten by malloc and free.** This means your pointers (the addresses) are in stack memory, but they're free to point to any memory region whether that memory region is in the stack or on the heap.

8.5 The sizeof Operator

Many times, you're too lazy to manually compute the number of bytes you need. For example, you know that you want enough space for an array of `int` with 12345 elements. Doing the math yourself is such a pain. Of course you can type malloc(4 * 12345); since you know that int takes 4 bytes. But in case you forget or you want to make the code more readable, we'll introduce a new operator called `sizeof`. Using this operator actually looks like a function call, but the compiler evaluates it while your code is compiling to compute sizes. Here is some sample code.

```
#include <malloc.h>

int main() {
  int* arr = (int*)malloc(sizeof(int) * 10000);
}
```

`sizeof` is also really useful for when you define our own custom data types. One day you might want to make our data type take 16 bytes, but then you update it so it needs 24 bytes. In that case, you don't

want to go through all our code to replace all the references of `malloc(16 * x);` with `malloc(24 * x);`. Instead we can just do `malloc(sizeof(custom_data_type) * x);`.

8.6 The Null Pointer

malloc makes a guarantee that it will never return an address below 4096 for modern operating systems. The value of 4096 changes based on computer specs and operating system, but it's guaranteed that it will never allocate you the memory address 0. So if you want to say that a pointer points to nowhere, set its value to 0. If you want to check if a pointer is uninitialized, check if it points to 0. This is a convention for everybody and is the universally accepted way of knowing whether a pointer is uninitialized.

8.7 Memory Errors

Lastly, here are some common errors you might encounter when playing around with memory.

8.7.1 Segmentation Fault

When you access a memory location that does not belong to you, you get something called a segmentation fault. Usually this happens in three scenarios. One, you allocate 100 bytes, but try to access the 101th byte. Two, you access memory that you have not even allocated yet. Three, you access memory that you have already freed. Sometimes, it's possible that your code continues on. But this kind of scenario is very bad.

8.7.2 Memory Leak

When you allocate memory, you are given a pointer to the address of the memory you requested. If you update your pointer to something else, say you request a new chunk of memory, and you forget to free the first block of memory, you will never again be able to recover that memory address. Since you can't access that memory address, that memory is lost forever until your program stops. This situation is called a memory leak. Some software and video games are notorious for this and they will gradually eat up more memory until you kill them. For programming contests, if you allocate, say, 100MB per test case but fail to free your memory, you will likely hit a memory limit exceeded error after a few cases.

9 Strings, Character Arrays, and Encoding

9.1 C Strings

Before this new `string` data type from C++, C only had character arrays. In C, a string is only a char array with the last char equal to 0.

To get the length of a string, you would loop through all characters until you hit 0. To add a character to the string, you just write one more character at the end, making sure that the new end is 0 again. It's all very simple. And if you have a C++ string, you can get a pointer to the original C string using `s.c_str()`.

The reason we bring this up is because we want to talk about how characters are represented in computers.

Computers only work with numbers. They know how to copy, add, subtract, and so on. But characters are not numbers. What we really need to do is to decide a mapping from each number to a character. Thankfully, other people have decided that mapping for us. The most important mapping of numbers to characters to learn is called ASCII. There are tables online for ASCII, but memorizing the table doesn't have very much use. It's enough to know that the mapping exists and that's computers will look up characters from the table to decide what to display on your screen.

By looking up some values on the table, we can learn that 20 maps to the space character. 10 maps to the new line character (the "enter" character). 65 maps to A. 97 maps to a. In general, ASCII maps numbers from the range 0 to 127 into characters, and this is enough to fit in one byte. The numbers from 128 to 255 vary wildly, so we won't go into that. The following code demonstrates my point about the equivalence of characters and numbers:

```
#include <stdio.h>

int main() {
  if ('A' == 65) {
    printf("'A' is equal to 65\n");
  }
  if ('z' == 122) {
    printf("'z' is equal to 122\n");
  }
  printf("Let's print the %c character\n", 65);
}
```

You might notice from this that the 'x' notation for characters is just convenience for letting the compiler look up the number for you during the compilation stage. There's not very much reason to look up the character yourself when the compiler can do it. The important thing is that the equivalence of letters and numbers allows you to do operations on characters. The ASCII table was designed for a purpose, so it's not just completely jumbled garbage. Digits are consecutive on that table, and so are capital letters and small letters. For example, if you wanted to capitalize letters, you just need to do some arithmetic to convert the range 97-122 ('a' to 'z') to 65-90 ('A' to 'Z').

This is practically all you'll need to know about character encoding for programming contests.

9.2 Sample String Upper Case

To demonstrate what I mean about using ASCII with arithmetic, here is some example code that capitalizes one line from the input.

```
#include <iostream>
using namespace std;

int main() {
  char string[1000];
  cin.getline(string, 1000);
  for (int i=0; i<1000; i++) {
    // Terminating character
    if (string[i] == 0) {
      break;
    }

    // if within the range of 'a' to 'z'
    if ('a' <= string[i] && string[i] <= 'z') {
      // Subtract 'a' to make it 0 to 25, then add 'A' to capitalize.
      string[i] = string[i] - 'a' + 'A';
    }

    // equivalent to the code above
    // if (97 <= string[i] && string[i] <= 122) {
    //   string[i] = string[i] - 97 + 65;
    // }
  }
  cout << string << endl;
}
```

10 C++ Structs

We have been mentioning defining our own data types for a while now. It's time we learn to do that now.

Let's start with a simple problem. We want to write a function that computes the simplest form of a fraction.

If you actually try to think of how to do this, you'll quickly run into a problem: Fractions are represented by a numerator and a denominator. But functions can only return one value. If you're willing to mess around, I'm sure some of you will be able to think of solutions to this question.

Using what you already know by now, I can imagine three tricks to get around this problem. The first way is to store return values in global variables and copy them over as soon as the function is done. A second way is to accept pointers and store the answers in the given addresses. And third, allocate some new memory, store the answer there, then return a pointer to that memory address. Any of these solutions will work, but they don't fit into the contrived story for this section! Not to mention it's going to be difficult to work with these things.

Wouldn't it be nice if we could just bundle variables together in a simple and elegant way? Then we could return those bundles so we'll be able to return two variables. And taking that idea further, we can pass those bundles as arguments to functions, saving us lots of typing and error-prone code. What a wonderful world that would be if only we could bundle variables together.

Well I have good news for you! C++ supports exactly that through a feature called a **struct**! Another name you might read on the internet for a struct is a record, which is the term used for the same thing in a few older programming languages (and even some newer ones like the Starcraft II modding language called Galaxy). To understand what a struct is, let's look at some sample code.

```
struct fraction {
  int numerator;
  int denominator;
};
```

The sample code above defines a struct called `fraction`. In this case, fraction is a bundle of two `int` variables. The variables that it groups together are called its members. In this case, our fraction struct has two members called numerator and denominator. Let's look at some sample code to see how we can use our new data type.

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

struct fraction {
  int numerator;
  int denominator;
};

int gcd(int a, int b) {
  // The GCD is the largest number both numbers are divisible by.
  int smaller = std::min(a, b);
  for (int i=smaller; i > 0; i--) {
    if (a % i == 0 && b % i == 0) {
      return i;
    }
  }
  return 1;
}

fraction simplify_fraction(fraction f) {
  // We simplify a fraction by dividing its numerator and denominator by their gcd.
  int g = gcd(f.numerator, f.denominator);
  fraction answer;
  answer.numerator = f.numerator / g;
  answer.denominator = f.denominator / g;
  return answer;
```

```
}

int main() {
  // create a new fraction called ‘unsimp‘
  fraction unsimp;
  unsimp.numerator = 15;
  unsimp.denominator = 20;

  // create a new fraction called ‘simp‘
  fraction simp = simplify_fraction(unsimp);

  printf("The unsimplified version is %d / %d\n", unsimp.numerator, unsimp.denominator);
  printf("The simplified version is %d / %d\n", simp.numerator, simp.denominator);
}
```

Looking at the sample code, you'll see that members are accessed via the `.` operator. You will also notice that we passed fraction objects around as if they were any other data type. We will eventually learn how to do operations on our custom operations.

In the sample code above, `f`, `unsimp`, `answer`, and `simp` are called **objects** of our new data type `fraction`. So a struct is a definition of a data type, and an object is an occurrence of that data type. Sometimes you might also encounter the word **instance** which means the same as object.

One common error: You need a semi-colon ; after the closing bracket } of the struct. Also you need a semi-colon ; at the end of each member declaration. So when you're defining structs and you suddenly get a syntax error, that's one of the most likely places to look.

One more thing I would like to note at this point is that for those of you who may have heard the term "data structure" before, I would like to distinguish a struct from a data structure. Some online references might confuse the two because of a similar-sounding name, but they are different. struct is only a keyword in C++ that allows you to group variables together. We will be covering actual data structures later on, so hold on to your hats!

10.1 The Arrow Operator

C++ has a special operator for pointers to structs. If you have a pointer to a struct, you can use `->` to access its members instead of having to dereference the pointer and use the `.` operator. In other words, `(*ptr).member` means the same thing as `ptr->member`. The arrow notation was introduced simply for convenience, and it's recommended to use because it looks nicer.

11 Homework

11.1 Linked List

Implement a linked list of `int` from scratch. In particular, fill out the commented sections of the code below to make it work. You are free to add any structs you want.

```
#include <iostream>
using namespace std;

struct linked_list {
   // Fill out
   int length;
};

linked_list* linked_list_create() {
   // Fill out
}
```

```
void linked_list_destroy(linked_list*) {
   // Fill out
   // Make sure you don't have memory leaks!
}

void linked_list_add(linked_list* list, int value) {
   // Add something to the end of the linked list
   // Fill out
}

void linked_list_remove(linked_list* list, int index) {
   // Fill out
}

int linked_list_get(linked_list* list, int index) {
   // Fill out
}

int main() {
   // Input is lines of the form:
   // add v
   // remove idx
   // get idx

   linked_list* list = linked_list_create();

   while(1) {
       string instr;
       cin >> instr;
       if (instr == "add") {
          int v;
          cin >> v;
          linked_list_add(list, value);
       } else if (instr == "remove) {
          int idx;
          cin >> idx;
          linked_list_remove(list, idx);
       } else if (instr == "get") {
          int idx;
          cin >> idx;
          cout << linked_list_get(list, idx) << endl;
       } else if (instr == "length") {
          cout << linked_list->length;
       } else if (instr == "exit") {
          break;
       }
   }
   linked_list_destroy(list);
}
```

11.2 Jagged Array

Subtask 1. Allocate a 2D array from the heap.

Subtask 2. Allocate a jagged array (i.e. a 2D array with each subarray not having equal size) from the heap. Row i should have space for (i+1) elements.

11.3 Problems

UVa 11278 - One-Handed Typist
UVa 554 - Caesar Cypher