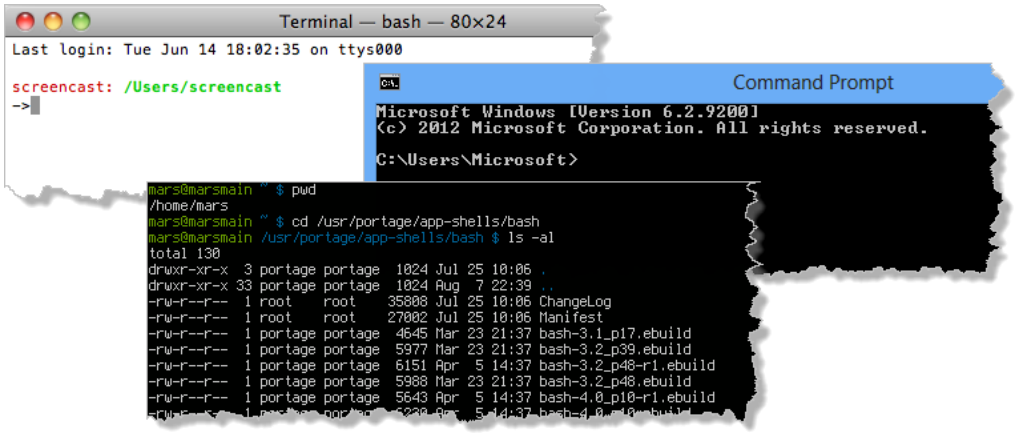
Chapter

23

The Command Line

Depending on your operating system and C++ development environment used, there are different methods of starting a program. All of these methods ultimately rely on the operating system's facility to **load and run a program**. While you can write C++ code which calls these facilities directly. Your IDE though, will normally make use of your operating system's **command processor**, also known as the **shell**.

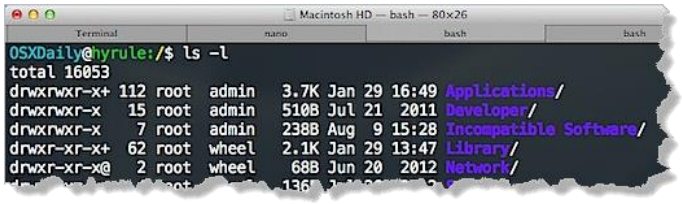
In Windows, the shell is usually the program named **`cmd.exe`**, although later versions of Windows come with a more powerful shell program called **`Powershell`**.



In Linux and on the Mac, the most popular shell program is **`bash`** (the Bourne-again Shell; a typical UNIX pun). That is the one we are using in our CS50 IDE workspace.

The Command Line

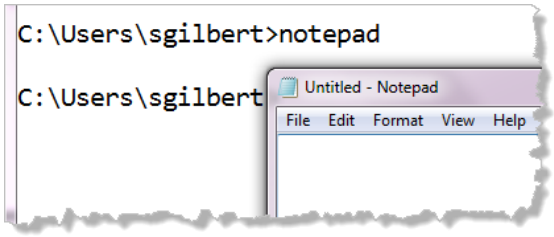
The command processor **accepts an instruction** (in text form), **interprets it**, and then converts the request into the **internal system calls** needed to carry out the action. Common shell commands **list** the files in a directory or **copy**, **move** and **delete** files.



The commands are often different for different shells. In Windows, you list files in the directory with **dir**, while in Linux and on the Mac, you use the **ls** command.

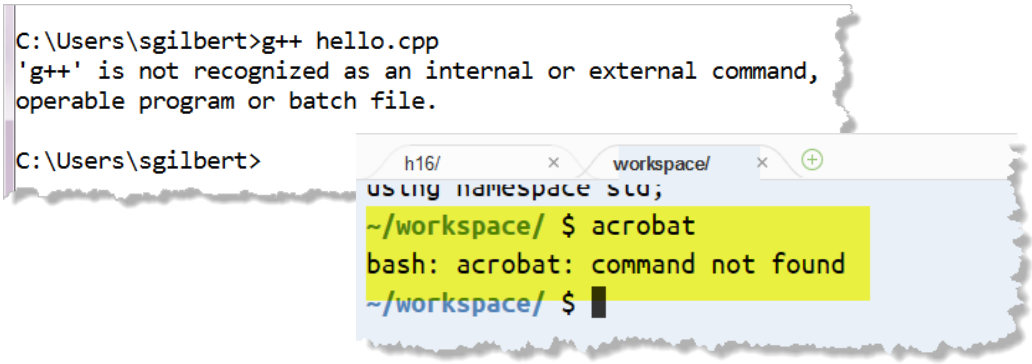
Running Programs

The most common shell command is to **launch a program**. Simply to **type the name of the program** into the shell, at the command prompt, like the picture shown here.



When you type this and press **ENTER** the command-processor first searches a special area of memory (called the **executable path**), looking for an executable program that matches this pattern. When it finds one (**notepad.exe**) it launches the program.

If it **cannot find the program (because it has not been added to the executable path or because it hasn't been instal**led on the machine you are running on), the command processor prints an error message.



> *On Windows, the current directory **is** placed in the executable path; on Linux and the Mac it is **not**; on those platforms, you have to type the current directory (**./**) before the name of the executable.*

Command-Line Arguments

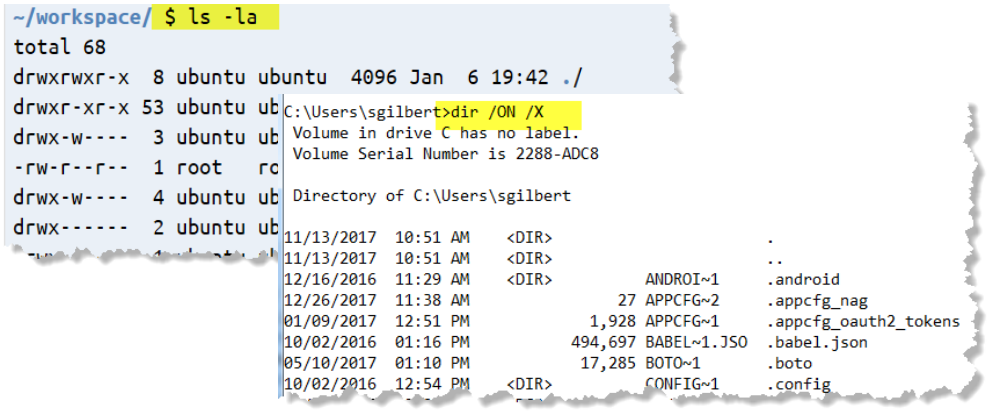
Instead of **prompting for input**, you can supply **additional information** on the command line. These additional strings are called **command line arguments.**

```
$ make hello
```

If you type this, the **make** program receives **"hello"** as a **command line argument**. It is up to the program what to do with it; **make** attempts to build a target named **hello**.

Command-line Switches

Strings starting with a hyphen (**-**) are customarily considered **options** (or **switches)** and other strings as file names. In Windows, such switches often start with a forward slash. In Linux, type **`ls -la`** and press **`ENTER`**. You can get a similar display in Windows by using **`dir /ON /X`**.



A New Version of main

To use command line arguments in your code, modify the signature of **`main()`** by adding two parameter variables: an integer and an array of C-strings of type **`char*`**.

```
int main(int argc, char *argv[])
{
    // your code here
}
```

The first parameter, (**`argc`**), is the **number of elements** in the command-line array. The second, (**`argv`**), is an array of **`char*`**, because each argument is passed as a character array or C-style string.

The first element, **`argv[0]`** is the name of the program, **as invoked on the command line**. A common error is to grab **`argv[0]`** when you want **`argv[1]`**. You need not use the names **`argc`** and **`argv`**, but it may confuse people if you don't.

Traversing the Arguments

This program prints out the command-line by stepping through the array.

Each whitespace-delimited cluster of characters on the command line is turned into a separate array argument. However, the rules for deciding what makes up a "cluster" are up to the operating system. Placing quotes around a pair of words works in any operating system; other features, like back-ticks, work only under some. Try some.

Converting Arguments

The command-line consists of C-style strings; to use any C++ **string** functions from the standard library, first **assign** the array element to a local **string** variable.

To treat an argument as a number, convert it with the **stoi()**, **stol()** and **stod()** function in the **<string>** header. **<cstdlib>** also includes helpers, **atoi()**, **atol()**, and **atof()**, which convert ASCII character arrays.

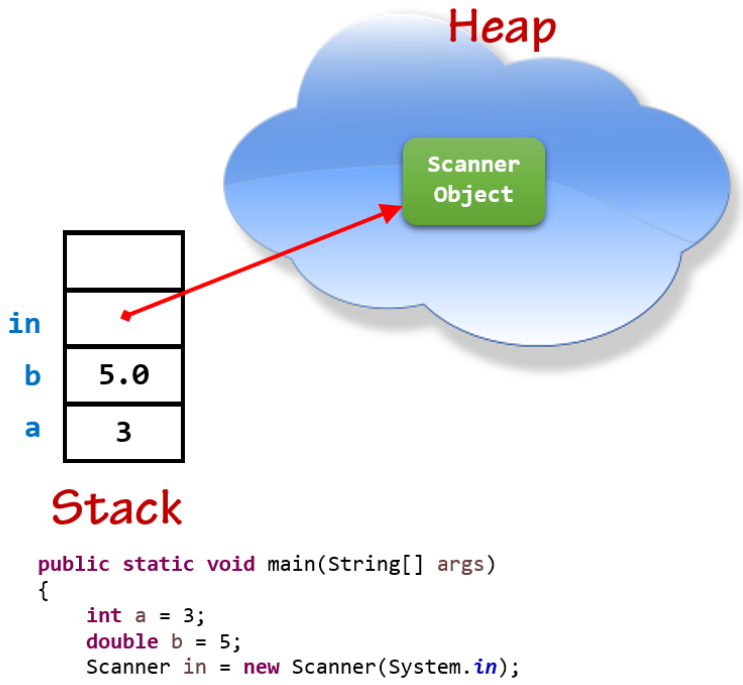
```
1  #include <iostream>
2  #include <cstdlib>
3  using namespace std;
4
5  int main(int argc, char* argv[])
6  {
7      for (int i = 1; i < argc; i++)
8          cout << atoi(argv[i]) << endl;
9  }
```

Converting arguments to numbers.

Notice that loop starts at **1** to skip over the program name at **argv[0]**. If you pass a floating-point number on the command line, **atoi()** takes only the digits up to the decimal point. If you pass non-numbers, these come back from **atoi()** as zero.

Dynamic Memory

You're already familiar with dynamic memory, since that's what is used to **create every object** in Java. Writing `new Scanner(System.in)`, allocates memory in the heap to store a new `Scanner` object as shown here.




```
public static void main(String[] args)
{
    int a = 3;
    double b = 5;
    Scanner in = new Scanner(System.in);
```

In C++, however, it is not enough to **allocate memory**; you also have to **free that memory** when it is no longer needed. The process of doing this in a disciplined way is **manual memory management**.

Allocating memory from the heap is common in programming. All the standard library collection classes use the heap to store their elements. In CS 250 and CS 200, you'll have many opportunities to build your own versions of these collection classes.

Before doing so, it is important to learn the underlying mechanics of dynamic allocation and how the process works.

The new Operator

Like Java, C++ uses the **new** operator to allocate memory on the heap. In its simplest form, the **new** operator takes **a type** and allocates space for **a single variable** of that type located on the heap. For example, look at this code:

```
int *p = new int;
```

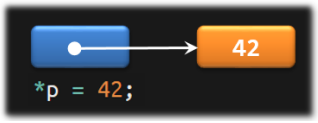
The **new** operator **returns the address** of the variable it allocates. I just ran this on my machine and the first location on the heap was located at address **0x1ef90**, so that is what is stored in the variable **p**.

Knowing the exact address value stored in **p** is really meaningless; instead, realize that whatever the address, the variable **p** on the stack **always points to** the newly allocated **int** on the heap. That means we can **indicate the relationship with an arrow.**

Dereferencing the Pointer

To use the variable, just **dereference the pointer**. To store **42** just write this:

```
*p = 42;
```



Initializing Heap Memory

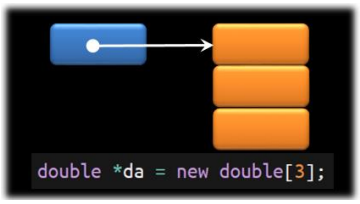
Using the raw **new** operator to create an object leaves the variable **uninitialized** if it is a primitive or built-in type and **default initialized** for class types. Instead creating **uninitialized** elements on the heap, you **initialize** individual variables by **using direct initialization**. In C++11, you can use uniform initialization.

```
int *p1 = new int;          // uninitialized
int *p2 = new int(42);      // direct initialized
string *p3 = new string;    // default initialized
double *p4 = new double{};  // default (c++11)
```

Dynamic Arrays

You can **create arrays on the heap**; these are called **dynamic arrays**. Follow the type name with the desired number of elements enclosed in square brackets:

```
double *da = new double[3];
```



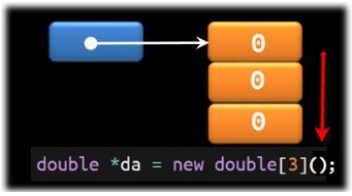
Here, **da** points to **the first double** in a block of memory large enough to hold three **double**s. Treat **da** as an array

whose storage **lives in the heap** rather than on the stack. Remember to **save the size of the array**, though, since, you cannot use the "**sizeof** trick" on a dynamic array.

Dynamic Array Initialization

Before C++ 11, elements of a dynamic array were **uninitialized**. With the new standard, you can value initialize (that is, **set to zero for primitive types**), **all the elements** on the heap by using a set of empty parentheses after the square brackets.

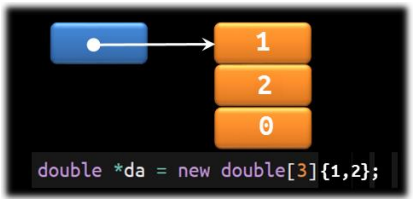
```
double *da = new double[3]();
```



Now, each element pointed to by the **da** pointer is guaranteed to hold **0.0**.

You can also use **list-style uniform initialization** to partly initialize a dynamic array:

```
double *da = new double[3]{1,2};
```

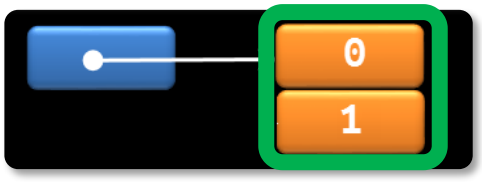


Dynamic Objects

The **new** operator can also allocate **objects or structures** on the heap. Let's assume you have a **Fraction** class. If you use only the class name, then C++ allocates space for a default **Fraction** object on the heap.

```
Fraction *fp = new Fraction;
```

If **Fraction** has a default constructor, the **new** operator automatically calls it, passing the address of the newly allocated memory. Assuming the constructor initializes each data member to the values **0/1**, you'll end up with something like this:



Supply arguments, and C++ will call the matching overloaded constructor.

Freeing Memory

Computer memory is finite, so the heap may eventually run out of space. When this occurs, the **new** operator throws a **bad_alloc** exception. There is usually nothing the program can do to recover. With a modern O/S and virtual memory, this is rare.

Unlike Java, C++ programmers must **manually free heap variables** when they are no longer used. In Java, this is handled by the **garbage collector**. You free a heap variable by using the **delete** operator like this:

```
Fraction *fp = new Fraction;
// Use the fraction object here
delete fp;  // free the memory
```

When you allocate **an array**, you add square brackets after the **delete** keyword.

```
double *da = new double[3];
// Use the array
delete[] da;    // free heap memory
```

Click on the link here to **visualize** several uses of pointers.

Finish Up

- Complete the **reading exercises (REX)** for this chapter.
- Complete the homework using the **CS50 IDE**. The link is on Canvas.
  a. Make sure you submit the assignment using **make submit**.
  b. Make sure you check the CS150 Homework Console to see that your scores got reported, before the beginning of the next lecture.
- Take the **pre-class reading quiz** on Canvas. You have two attempts.

See you in class or on the Canvas discussion board.