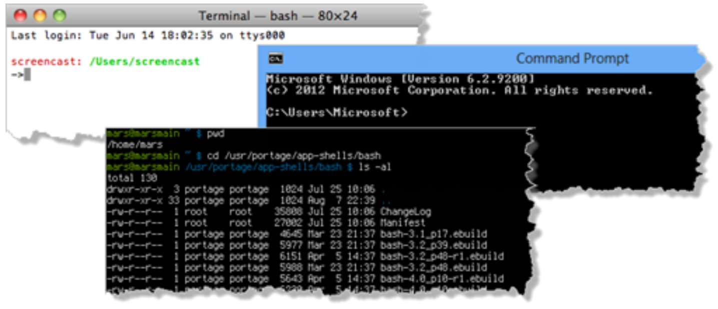
The Shell

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 will normally make use of your operating system's **command processor**, also known as the **shell**.



```
Last login: Tue Jun 14 18:02:35 on ttys000

screencast: /Users/screencast
->
```

```
Microsoft Windows [Version 6.2.9200]
(c) 2012 Microsoft Corporation. All rights reserved.

C:\Users\Microsoft>
```

```
mars@marsmain ~ $ pwd
/home/mars
mars@marsmain ~ $ cd /usr/portage/app-shells/bash
mars@marsmain /usr/portage/app-shells/bash $ ls -al
total 130
drwxr-xr-x  3 portage portage  1024 Jul 25 10:06 .
drwxr-xr-x 33 portage portage  1024 Aug  7 22:39 ..
-rw-r--r--  1 root    root    35808 Jul 25 10:06 ChangeLog
-rw-r--r--  1 root    root    27002 Jul 25 10:06 Manifest
-rw-r--r--  1 portage portage  4645 Mar 23 21:37 bash-3.1_p17.ebuild
-rw-r--r--  1 portage portage  5977 Mar 23 21:37 bash-3.2_p39.ebuild
-rw-r--r--  1 portage portage  6151 Apr  5 14:37 bash-3.2_p48-r1.ebuild
-rw-r--r--  1 portage portage  5988 Mar 23 21:37 bash-3.2_p48.ebuild
-rw-r--r--  1 portage portage  5643 Apr  5 14:37 bash-4.0_p10-r1.ebuild
```

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 workspaces.

Command Line Arguments

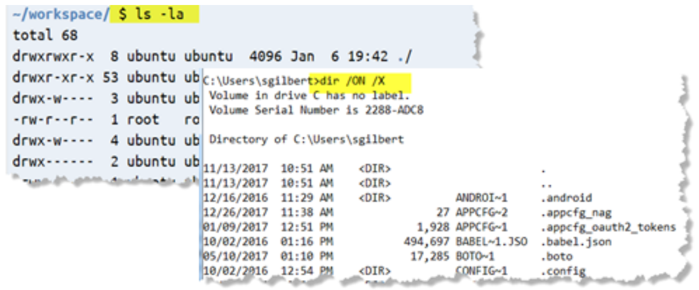
Instead of prompting for input inside your program, you can supply additional information on the command line. These additional strings are called **command line arguments**.

Consider the `make` program. If you type this:

```
$ make hello
```

the `make` program receives `"hello"` as a **command line argument**. It is up to the program to decide what to do with it; make attempts to build a target named `hello`.

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, when you type `ls -la` and press `ENTER`, you will get a similar display as if you typed `dir /ON /X` on Windows.



```
~/workspace/ $ ls -la
total 68
drwxrwxr-x  8 ubuntu ubuntu  4096 Jan  6 19:42 ./
drwxr-xr-x 53 ubuntu ub
drwx-w----  3 ubuntu ub
-rw-r--r--  1 root   ro
drwx-w----  4 ubuntu ub
drwx------  2 ubuntu ub
```

```
C:\Users\sgilbert>dir /ON /X
 Volume in drive C has no label.
 Volume Serial Number is 2288-ADC8

 Directory of C:\Users\sgilbert

11/13/2017  10:51 AM    <DIR>                       .
11/13/2017  10:51 AM    <DIR>                       ..
12/16/2016  11:29 AM    <DIR>          ANDROI~1     .android
12/26/2017  11:38 AM                27 APPCFG~2     .appcfg_nag
01/09/2017  12:51 PM             1,928 APPCFG~1     .appcfg_oauth2_tokens
10/02/2016  01:16 PM           494,697 BABEL~1.JSO  .babel.json
05/10/2017  01:10 PM            17,285 BOTO~1       .boto
10/02/2016  12:54 PM    <DIR>          CONFIG~1     .config
```

Using Command Line Arguments

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

```
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 really wanted the first argument, which is `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.

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

int main(int argc, char *argv[])
{
    cout << "The program name is " << argv[0] << endl;
    for (int i = 1; i < argc; ++i)
    {
        cout << "Argument " << i << " is " << argv[i] << endl;
    }
    return 0;
}
```

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 (`argv[i]`) to a local `string` variable. Once you've converted the C-string argument to a C++ `string`, you may then treat the argument as a number by converting it with the `stoi()`, `stol()` and `stod()` function in the `<string>` header.

However, you **may** convert directly from C-style strings to numbers, with out first converting to `string`, by using the helper functions in `<cstdlib>` which include `atoi()`, `atol()`, and `atof()`.

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

int main(int argc, char *argv[])
{
    for (int i = 1; i < argc; ++i)
        cout << atoi(argv[i]) << endl;
}
```

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.

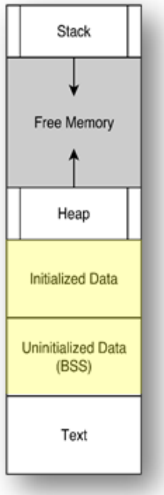
Introducing the Heap

The illustration on the right shows how memory is used when your program runs. **Global** variables (and constants) are placed in the **static storage area** (along with the program's **code or text**), when your program is loaded. These remain **in the same location** while your program is running. This is called **static allocation**.

Local variables and parameters are created **when encountered** during **runtime**. These are **allocated on the stack**. When you call a function, memory is allocated for the variable; when the function returns, that memory is freed. This is known as **automatic allocation**.

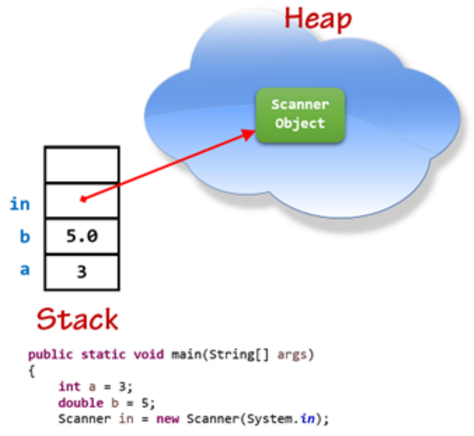
Finally, **explicitly requesting** memory from the operating system as the program runs is called **dynamic allocation**. The memory for these dynamic variables is **allocated on the freestore or heap**.

This makes it possible for data structures to expand as your program runs. Classes like `vector` and `string` depend on this; when a `vector` needs more memory, for a `push_back()` say, it requests it using dynamic allocation.



Stack vs. Heap

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);
```

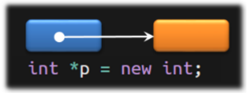
In C++, however, it is not enough to **allocate memory**; you also have to **free that memory** when it is no longer needed. The process of doing this in a disciplined way is called **manual memory management**.

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;
```

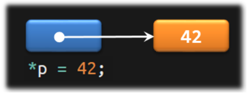


The `new` operator **returns the address** of the variable it allocates. I just ran this on my machine and the first location on the heap was located at **address `0x558f24791eb0`**, so that is what is stored in the variable `p`.

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.**

To use the object on the heap, just **dereference the pointer**. To store `42` just write this:

```
*p = 42;
```



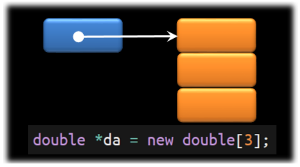
Using the raw `new` operator to create an object leaves the variable **uninitialized** if it is a primitive or built-in type and **default initialized** for class types. Instead creating **uninitialized** elements on the heap, you can **initialize** individual variables by **using direct initialization**. In C++11, you can also 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 also create arrays on the heap; these are called dynamic arrays.
Follow the type name with the desired number of elements enclosed in square brackets:

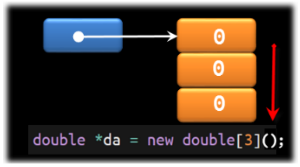
```
double *da = new double[3];
```



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.

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.

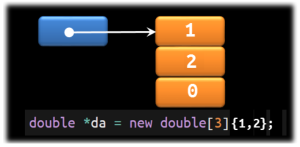
```
double *da = new double[3]();
```



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};
```

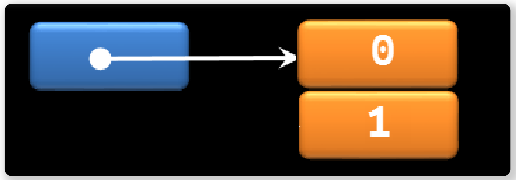


Dynamic Objects

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 very 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**. In C++ 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
```

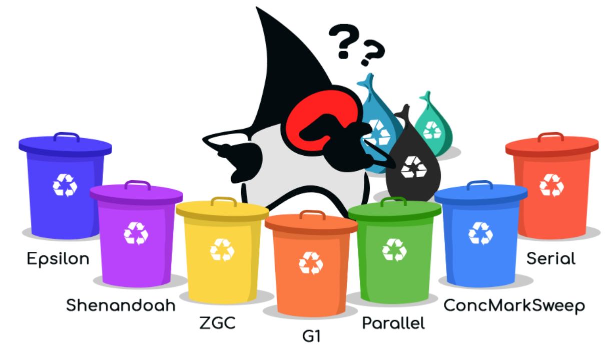
When you allocate **an array**, you add square brackets after the `delete` keyword, like this:

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

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

Memory Management

Although Java uses the `new` operator, just like C++, it has no corresponding **`delete`** operator. That's because in Java, when you run your program, a set of smaller programs **run at the same time**, scouring the heap, looking for unused objects, and **automatically** freeing them. This automatic memory management in Java is called **garbage collection**.



In contrast, C++ manual memory management, using **raw pointers** with **`new`** and **`delete`**, requires an almost superhuman attention to detail. Mistakes will cause your program to leak memory, or, to corrupt the memory manager itself.

Let's look at the three most common pitfalls that accompany manual memory management: the **memory leak**, the **dangling pointer**, and the **double `delete`**. Then, you'll meet C++11's new **smart pointers**, which ameliorate many of the failings of pure manual memory management.

Memory Leaks

The `delete` operator frees the allocated memory on the heap, but it does not physically change the pointer in any way. This leads to three possible errors. Here is one:

```
bool validDate(int yr, int mo, int da)
{
    Date *pd = new Date(yr, mo, da);
    if (! pd->isValid()) { return false; }
    cout << "Date " << (*pd) << " OK" << endl;
    delete pd;  // free heap memory
    return true;
}
```

This function constructs a new `Date` object **on the heap**, and then calls the `isValid()` member function to see if the resulting combination is legal. If it is **not valid**, the function returns `false`. If it is valid, then the function prints the `Date` object, deletes the object on the heap and returns true.

This code has **one `new`** and **one matching `delete`**, but it still **has a memory leak** whenever an invalid data is entered. That's because there are two `return` statements in the function. If the `Date` is invalid, the function returns **without deleting the data on the heap**.

Try running the program oneline.

Dangling Pointers

A dangling pointer is a pointer that contains an address, but the address points to data you have already deleted. Here's a function for a rather strange contest:

```cpp
bool hasWon(int y, int m, int d)
{
    Date *pd = new Date(y, m, d);
    delete pd;  // avoid leaking
    return pd->isValid() && pd->year() % pd->day() == 0;
}
```

The `Date` object allocated on the heap is **deleted** before the function returns. But, **after the `Date` has been deleted**, the pointer is used to call three functions. At this point, **`pd` is a dangling pointer**, which is a pointer to heap memory that you **no longer "own"**.

The insidious thing is that the function will do exactly what you want, but only under some circumstances. If you run the link, you'll see that since the `Date` on the heap hasn't changed, when you call the functions they still work correctly. This is similar to checking out of a hotel, but keeping a copy of the key. You may be able to stay another night for free, if the place isn't too busy, and no one catches you, but you **might** get into trouble.

Of course, the code is still illegal and won't work at all on some platforms. For instance, if you run the same code on Codespaces, you'll see it doesn't produce the right answer. However, it also **doesn't provide you with any indication that you've made a mistake**.

To help you find your mistakes, you can set the pointer to `nullptr` every time you delete the pointer. Here's a template function you can use in place of `delete` that will do that for you.

```cpp
template <typename T>
void delete_raw(T& p) {
  delete p;
  p = nullptr;
}
```

Double Deletes

When you discover a memory leak, your first inclination is to start adding `delete` statements. That can create more problems. Here is an example:

```
void checkDate(int yr, int mo, int da)
{
    Date *pd = new Date(yr, mo, da);
    if (! pd->isValid()) delete pd;     // avoid leak
    else cout << (*pd) << " is OK" << endl;
    delete pd;  // OOPS, a double delete
}
```

Here, the programmer adds a `delete` for **both** the `true` and `false` branches (unlike the memory leak example, where an invalid `Data` was never **deleted**. However, because the programmer **forgot** to add braces around the two statements in the `else` part, the final `delete` is a **double `delete`: deleting an already freed pointer** whenever the `Date` is invalid.

This is also a big no-no. **This should always result in a runtime error**, because this is almost certain to corrupt the heap. Technically, however, it results in **undefined behavior.**