Memory is just a sequence of byte-sized storage devices. The bytes are assigned numeric addresses, starting with zero, just like the indexing of the cells of an array.

It is the job of the operating system (OS) to:

- manage the allocation of memory to processes
- keep track of what particular addresses each process is allowed to access, and how
- reserve portions of memory exclusively for use by the OS
- enforce protection of the memory space of each process, and of the OS itself
- do all this as efficiently as possible
When a function call occurs, storage space must be available for:

- parameters that are passed by value
- local variables declared within the function
- the return value, if any

This is accomplished by creating a data object, called an *activation record*, whenever a function call takes place. The activation record contains storage space for all of the items mentioned above, and perhaps more.

When a function terminates, the corresponding activation record is destroyed.

It is natural to organize these activation records on a stack. Why?

Each process has such a stack, maintained by the system as the process runs.

The process cannot directly manipulate the stack, but it is allowed to access those portions of it that correspond to its local variables.
Processes also often need to create data objects "on the fly" as they execute.

This is accomplished by making a call to the OS requesting that the necessary amount of memory be allocated to the process.

The OS responds by either:

- returning the address of the first byte of a chunk of memory allocated to the process
- denying the request

Dynamically-allocated memory is allocated from a reserved region of system memory called the *heap*.
**Pointers**

**Pointer Concepts and Syntax**

- **pointer** a variable whose value is a memory address
- **pointee** a value in memory whose address is stored in a pointer; we say the pointee is the *target* of the pointer

Since memory addresses are essentially just integer values, pointers are the same width as integers.

A pointer has a type, which is related to the type of its target.

Pointer types are simple; there is no automatic initialization.

A pointer may or may not have a target.

Given a pointer that has a target, the target may be accessed by *dereferencing* the pointer.

A pointee may be the target of more than one pointer at the same time.

Pointers may be assigned and compared for equality, using the usual operators.

Pointers may also be manipulated by incrementing and decrementing, although doing so is only safe under precisely-defined circumstances.

By convention, pointers without targets should be set to 0 (or NULL).
## Basic Pointer Syntax

### Declarations:

```c
int x = 42, y = 99;
string s = "Pointers are good for you!";

int* p1;  // declaration of pointer-to-int
string* p2;  // pointer-to-string
```

### Setting targets:

```c
p1 = &x;  // p1 points to x
p2 = &s;  // p2 points to s
```

### Dereferencing:

```c
*p1 = 23;  // now x == 23
cout << *p2;  // prints string
```

```c
p1 = &y;  // now p1 points to y
p1 = NULL;  // now p1 has no target, AND
             // we can check that
```
Dynamic Allocation

The previous example involved only targets that were declared as local variables.

For serious development, we must also be able to create variables dynamically, as the program executes.

In C++, this is accomplished via the operator `new`:

```c++
int* p1 = new int;   // target is an uninitialized integer
int* p2 = new string; // target is a default string object
int* p3 = new string("I'm a target.");
```

The operator `new` performs a call to the underlying operating system requesting that memory be allocated. The amount of the request is implied by the parameters to `new`.

The result is either a returned address or a `bad_alloc` exception.
One of the most glaring differences between Java and C++ is how memory deallocation is accomplished.

In C++, we have static allocations, local or automatic allocations, and dynamic allocations. The first two are of no particular interest here.

Everything that your C++ program allocates dynamically must eventually be deallocated. The responsibility is yours.

Failure to deallocate memory in a timely but safe manner is one of the most common programming mistakes in most languages, including C++.

Deallocation is accomplished by using the operator `delete`:

```cpp
string *p1 = new string("I'm a target.");

// ...

delete p1;
```

`delete` does not reset the value of the pointer on which it is invoked!
It's important to understand just what `delete` does (and does not do).

First, `delete` can only be applied to a pointer storing the address of a target that was allocated by calling `new`.

Second, `delete` can only be applied to a pointer whose a target that has not already been deallocated.

Third, when `delete` is invoked on a pointer, the pointer is not automatically reset to `NULL`.

Fourth, `delete` causes the deallocation of the target of the pointer, not the deallocation of the pointer itself. You don't delete a pointer, you delete its target.
Arrays can be allocated dynamically, which sidesteps one of the fundamental limitations:

```cpp
double *p1 = new double[1000];
string *p2 = new string[42];
```

For simple types, there is no automatic initialization of the array cells.

For class types, each cell is initialized using the default constructor for the class.

The cells of the array can be accessed by using the pointer as the array name and the usual indexing syntax.

Deallocation of arrays is accomplished by using the operator `delete[]`:

```cpp
delete [] p1;
```
A pointer to an object can be used to access the public elements of the object:

```cpp
string *p1 = new string("I'm a target.");

cout << (*p1).length();  // parens are necessary

cout << p1->length();    // operator->() is alternative
```
It is possible, but frequently inadvisable, to have two or more pointers with the same target:

```cpp
string *p1 = NULL;
string *p2 = NULL;
p1 = new string("Target");
p2 = p1;
```

The basic issues are:

- *ownership*; which pointer is viewed as representing the primary "home" for the target object? This is generally where the responsibility for deallocation will lie.

- *dangling pointer*; if the target object is deallocated, then we may be left with pointers that store the address of a non-existent object

The C++ language provides little in the way of automated solutions.

The responsibility for managing these issues lies with the programmer.
Pointers Without Targets

There are a number of ways to create a pointer that has no (valid) target:

```c
int *p1;    // pointer has random value, dangerous!

int *p2 = NULL;    // no target, but NULL provides for a check

int *p3 = new int(42);    // find, p3 has a target

delete p3;    // p3 has no target; p3 is not NULL
```

A sensible principle would seem to be to set any pointer that does not have a target to `NULL`.

But, note this is unnecessary (and therefore a waste of execution time) in situations where the pointer cannot subsequently be used.
Does a Pointer Have a Target?

If a pointer is `NULL`, we know it has no target.

If a pointer is not `NULL`, there is, in fact, no general technique for detecting whether a given pointer does have a target.

It may be possible to query the operating system itself to determine whether a program actually owns a particular address:

```c
int *p1;
...
if ( OSSaysIOwnThisAddress( p1 ) ) {
  ...
}
```

Unfortunately, the hypothetical function call above would be specific to the particular OS installed on the system. Nothing in the Standard Library will do this.
What happens if a program dereferences a pointer which does not have a valid target?

The answer depends on the underlying OS.
- a poorly-designed OS may allow programs to access addresses they do not own
- such an error may lead to the throwing of an exception (Windows XP)
- such an error may lead to a signal from the OS (segfault in UNIX, Linux)

It is possible to catch exceptions thrown by Windows.

It is possible to write a signal handler that will be invoked automatically when a segmentation fault signal is sent on a UNIX system.

It is more efficient to design a system in which no "bad" pointer is ever dereferenced.
This is not easy, nor is it impossible.
Memory Leaks

We say that a *memory leak* occurs when a process loses access to dynamically-allocated memory before that memory has been deallocated:

```c
int *p1 = new int(42);
p1 = NULL;
```

When a process terminates, all its resources should be reclaimed by the OS.

However, in the situation above, the process will retain the allocation of the integer-sized chunk of memory until it terminates, even though the process has no way to make further use of that memory.

This is wasteful, and should be avoided.

Again, this is the responsibility of the programmer.
One cardinal novice sin is to design a function that returns a pointer to a local object:

```c
string* gimmeaString() {
    string localStringObject;  // ceases to exist on fn return
    return &localStringObject;
}
```

// caller:
```c
string *p = gimmeaString();
// p does not have a valid target
```
Example: Array Resizing

```c
int *p1 = new int[100]; // create array and use it awhile
...
int *p2 = new int[200]; // create larger array to replace it

for (unsigned int pos = 0; pos < 100; pos++) // copy values
    p2[pos] = p1[pos];

delete [] p1; // deallocate original array

p1 = p2; // reset original array pointer to new array

p2 = NULL; // eliminate alias
```
Equality vs Identity

\[x \text{ equals } y\]
x and y, in some precise sense, have the same value
In C++, this is equivalent to \(x == y\).

\[x \text{ is identical to } y\]
x and y are actually the same object
In C++, this is equivalent to \&x == \&y.

Side notes:
If x and y are pointers, then x equals y if and only if x and y have the same target.
In other words, two pointers are equal if and only if their targets are identical.
Pointers as Parameters

Passing a pointer to a function gives the function access to the target of the pointer. In effect, this implies another protocol for parameter passing: *pass-by-pointer*.

```c
void capitalizeString( string* Str ) {
    if ( Str == NULL ) return;
    for (unsigned int Pos = 0; Pos < Str->length(); Pos++) {
        Str->at(Pos) = toupper(Str->at(Pos));
    }
}
```

Here, the pointer is passed by value (which is the default), but its actual target may be modified by the function.

Similar to pass-by-reference, but we must use explicit pointer syntax in the function.
The name of an array is essentially a pointer to the zeroth cell of the array. So, when you pass an array as a parameter you are effectively passing the array by pointer.

However, there is one fundamental difference. If you make the declaration

```c
double Weights[100];
```
then you cannot subsequently modify the value of the array name. So, an array name is more like a pointer that is constant.

If you allocate an array dynamically, using a pointer variable, you can then use the usual array indexing syntax with the pointer:

```c
double *Weights = new double[100];
Weights[17] = 42.73;
```
Increment/decrement operations are allowed on C++ pointers. However, this is safe only if the target of the pointer is a cell of an array.

```cpp
void zeroArray( int* List, unsigned int Sz ) {
    if ( List == NULL ) return;
    int *Curr = List;
    for (unsigned int Pos = 0; Pos < Sz; Pos++, Curr++) {
        *Curr = 0;
    }
}
```

There's not much reason to increment the pointer here instead of simply using the loop counter as an array index.
From B. Stroustrup, “The C++ Programming Language”:

The result of applying the arithmetic operators +, −, ++, or -- to pointers depends on the type of the pointed to target object.

When an arithmetic operator is applied to a pointer \( p \) of type \( T^* \), \( p \) is assumed to point to an element of an array of objects of type \( T \); \( p+1 \) points to the next element of that array, and \( p-1 \) points to the previous element.

This implies that the integer value of \( p+1 \) will be \texttt{sizeof}(T) larger than the integer value of \( p \).
You can use pointer arithmetic to examine the contents of an object:

```cpp
string A("To be or not to be?");
char Byte = ' ';
char *p = (char*) &A;

for (unsigned int nByte = 0; nByte < sizeof(string); nByte++) {
    Byte = *p;
    cout << setw(3) << nByte << " : " << (int) Byte << endl;
    ++p;
}
```

You might be surprised by the results of this…
The combination of `const` with pointers raises some interesting capabilities:

```c
int* p = new int(5);        // nothing is constant
const int* p = new int(5);  // p is a pointer to an int which is constant
int* const p = new int(5);  // p is a constant pointer to an int (which may be modified)
const int* const p = new int(5); // p is a constant pointer to an int, which is also constant
```

Used sensibly, `const` can prevent many problems. For example:

```c
void zeroArray( int* const List, unsigned int Sz ) {
  
  
}
```
Besides pointer variables, C++ also has reference variables. Reference variables:

- store the address of an object (like a pointer)
- cannot be assigned a value after they are declared (like a `const` pointer)
- cannot ever be set to `NULL` (unlike a pointer)
- are implicitly dereferenced with no special syntax (unlike a pointer)
- cannot be targeted by `delete` (unlike a pointer)

```cpp
string Quote("To be or not to be...");
string &refToQuote = Quote;
cout << refToQuote.length() << endl
    << refToQuote << endl;
```

References are somewhat less flexible, and hence somewhat safer than pointers. However, it is still possible to have a reference that has no target.
References as Class Members

References are often useful as class members, but how can you set the value of the reference (since it can't be changed after it is created)?

class Log {
private:
    ostream& mFile;  // client-specified file for logging
public:
    Log(ostream& logFile);
};

Use the initializer list:

Log::Log(ostream& logFile) : mFile(logFile) {
}