Memory and Addresses

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

Run-time Stack

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.

---

**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

<table>
<thead>
<tr>
<th>Declarations:</th>
<th>Setting targets:</th>
<th>Dereferencing:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int* p1;</code></td>
<td><code>p1 = &amp;x;</code></td>
<td><code>*p1 = 23;</code></td>
</tr>
<tr>
<td><code>string* p2;</code></td>
<td><code>p2 = &amp;s;</code></td>
<td><code>cout &lt;&lt; *p2;</code></td>
</tr>
</tbody>
</table>

\[
\text{int } x = 42, \ y = 99; \\
\text{string } s = \text{"Pointers are good for you!";}
\]

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`:

\[
\text{int* p1 = new int;} \quad \text{// target is an uninitialized integer} \\
\text{int* p2 = new string;} \quad \text{// target is a default string object} \\
\text{int* p3 = new string("I'm a target.");}
\]

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.
Deallocation

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`:

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

`delete` does not reset the value of the pointer on which it is invoked!

C++ delete

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.
Dynamic Arrays in C++

Arrays can be allocated dynamically, which sidesteps one of the fundamental limitations:

```cpp
int *p1 = new double[1000];
int *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;
```

Pointers to Objects

A pointer to an object can be used to access the public elements of the object:

```cpp
int *p1 = new string("I'm a target.");
cout << (*p1).length(); // parens are necessary
cout << p1->length();   // operator->() is alternative
```
**Pointer Aliasing**

It is possible, but frequently inadvisable, to have two or more pointers with the same target:

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

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

int *p2 = NULL;  // no target, but NULL provides 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.

### "Bad" Pointers

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:

```
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.

Returning a Pointer to a Local

One cardinal novice sin is to design a function that returns a pointer to a local object:

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

// caller:
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
```