Lecture Summary

In this lecture, we begin a brief review of some of the basics, by talking about dynamic memory allocation. Being able to handle dynamically allocated memory and pointers (and recursion, the topic of the next lecture) are the key points to most data structures, and they are essential to object-oriented design in C++ as well.

1 The Basics: How Memory and Its Allocation work

At the physical level, computer memory consists of a large number of flip flops. Each flip flop consists of a few transistors, and is capable of storing one bit. Individual flip flops are addressable by a unique identifier, so we can read and overwrite them. Thus, conceptually, we can think of all of our computer's memory as just one giant array of bits that we can read and write.

Since as humans, we are not that good at doing all of our thinking and arithmetic in bits, we group them into larger groups, which together can be used to represent numbers. 8 bits are called 1 byte; beyond bytes, there are words (which are sometimes 16, sometimes 32 bits). Thus, more often, we conceptually regard our computer’s memory as one large (size $2^{32}$ or so) array of bytes. A lot of things are stored in this memory.

1. All variables and other data used by all programs.
2. But also the programs’ code, including the operating system’s.

The compiler and the operating system work together to take care of most memory management for you, but it is instructive to see what is going on under the hood.

When you compile your code, the compiler can examine primitive data types and calculate how much memory they will need ahead of time. The required amount is then allocated to the program in the stack space. For example, consider the following declarations:

```c
int n;
int x[10];
double m;
```

The compiler can immediately see that the code requires $4 + 4 \times 10 + 8 = 52$ bytes.

It will insert code that will interact with the operating system to request the necessary number of bytes on the stack for your variables to be stored. In the example above, the compiler knows that the memory will look as follows:

It knows the exact memory address of each variable; in fact, whenever we write `n`, this gets translated into something like “memory address 4127963” internally.

Notice that if we attempted to access `x[10]` here, we would access the data associated with `m`, and may end up reading (or overwriting) some of its bits. This would almost certainly have very undesired consequences for the rest of the program.

---

1. This space is called the stack space because as functions get called, their memory gets added on top of existing memory. As they terminate, they are removed in a LIFO (last in, first out) order.
2. That’s with the current sizes for integers and doubles. About 20 years ago, `int` were typically 2 bytes, and `double` 4 bytes. Your code should never have to depend on what is at this moment the size of the basic data types.
When functions call other functions, each function gets its own chunk of stack when it is called; it keeps all its local variables there, but also a program counter that remembers where in its execution it was. When the function finishes, its memory block is made available for other purposes again. Again, for statically allocated variables, the compiler can take care of all of this.

2 Dynamic allocation

Unfortunately, things aren’t quite as easy when we don’t know at compile time how much memory a variable will need. Suppose we want to do something like the following:

```cpp
int n;
cin >> n;
// create an array a of n integers
```

Here, at compile time, the compiler does not know how much memory the array will need. It can therefore not allocate room for a variable on the stack. Instead, our program needs to explicitly ask the operating system for the right amount of space at run-time. This memory is assigned from the heap.

The difference between static and dynamic memory allocation is summarized in the following table. To fully understand how dynamic memory allocation works, we need to spend some time on pointers.

<table>
<thead>
<tr>
<th>Static allocation</th>
<th>Dynamic allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size must be known at compile time</td>
<td>Size may be unknown at compile time</td>
</tr>
<tr>
<td>Performed at compile time</td>
<td>Performed at run time</td>
</tr>
<tr>
<td>Assigned to the stack</td>
<td>Assigned to the heap</td>
</tr>
<tr>
<td>First in last out</td>
<td>No particular order of assignment</td>
</tr>
</tbody>
</table>

Table 1: Differences between statically and dynamically allocated memory.

3 Pointers

A pointer is an “integer” that points to a location in memory — specifically, it is an address of a byte.

In C/C++, pointer types are declared by placing a star `*` behind a regular type name. Thus, `int *p; char *q; int **b; void *v;` all declare pointers. In principle, all these are just addresses of some memory location, and C/C++ does not care what we store there. Declaring them with a type (such as `int`) is mostly for the programmer’s benefit: it may prevent us from messing up the use of the data stored in the location. It also affects the way some arithmetic on memory locations is done, which we explain below.

Two of the ways in which “regular” variables and pointers often interact are the following:

---

3 This is called the heap space since it can be selected from any portion of the space that has not been allocated already. While the stack remains nicely organized, memory in the heap tends to be more messy and all over the place. Hence the name.

4 This means that all pointers are of the same size, and that the size of pointer used by a computer places a limit on the size of memory it can address. For example, a computer using typical 32-bit pointers can only use up to $2^{32}$ bytes or 4 gigabytes of memory. The modern shift to 64-bit architectures turns this to $2^{64}$ bytes, which will be enough for a while.
1. We want to find out where in memory a variable resides, i.e., get the pointer to that variable's location.

2. We want to treat the location a pointer points to as a variable, i.e., access the data at that location, by reading it or overwriting it.

The following piece of code illustrates some of these, as well as pitfalls we might run into.

```cpp
int* p, *q; //Pointers to two integers
int i,j;
i = 5; j = 10;
p = &i; //Obtain the address of i and save it to p
cout<<p; //Prints the value of p (address of i)
cout<<*p; //Prints the value of the integer that p points to (which is the value of i)
*p = j; // Overwrites the value of the location that p points to (so, i) with the value of j
*q = *p; // Overwrites the value of the location that q points to with the one that p points to
q = p; // Overwrites the pointer p with q, so they now point to the same location
```

A few things are worth noting here.

1. The last two commands both result in *p being equal to *q, but in different ways. In the first case, we copy the value from one location to the other, while in the second, we make both point to the same place.

2. The second-to-last command is very risky, and will likely cause a run-time error. We have not initialized q, so it points to an arbitrary memory location, quite possibly location 0. We are trying to overwrite it, which most likely the program is not allowed to do. The location may belong to the operating system or some other program. 

3. NULL is a special pointer we use for an uninitialized pointer variable, or to express that a pointer is not pointing anywhere. If we try to write *p when p == NULL, we get a runtime error.

4. & and * are inverses of each other. Thus, &*p and *&p are the same as p. (The exception is that &*p throws a runtime error when applied to p==NULL, instead of being equal to p.)

In general, it is quite easy to mess up with pointers. A few rules of thumb (which we discuss in class) will weed out common mistakes, but generally speaking, it is easy to forget to assign a pointer correctly, resulting in run-time errors.

We discussed before that pointers are basically just integers. They differ in that arithmetic is somewhat different. When you have a pointer p to an int, and you write p+1, the compiler assumes that what you want is the address where the next integer will start, which is 4 bytes later. Thus, the actual address referenced by writing p+1 is actually 4 bytes after the address of p.

This is where the type of pointer matters, as we hinted at above. When you have a void*, then addition really does refer to adding individual bytes. For all others, when you write p+k for a pointer p and integer k, this references memory location p+k*size(<type>), where <type> is the type of the pointer p.

## 4 Dynamic Memory Allocation

In order to dynamically allocate and deallocate memory, there are two pairs of functions, one in C style and one in C++ style. In C, the function for allocating memory is malloc, and for deallocation free. In C++, the functions are new and delete. We will first discuss the C style, as it is a little closer to the actual low-level implementation; we'll then see the C++ style, which shields us from some of the low-level details.

---

5 Fortunately, nowadays, all that happens is that your program throws a run-time error. In the past, the OS would often allow you to do such an overwrite, in which case often some complete system crashes would happen.
4.1 C Style

The function `void* malloc (unsigned int size)` requests `size` bytes of memory from the operating system, and returns the pointer to that location as a result. If for some reason, the OS failed to allocate the memory (e.g., there was not enough memory available), `NULL` is returned instead. The function `void free (void* pointer)` releases the memory located at `pointer` for reusing. A solution to our earlier problem of a dynamically sized array could look as follows:

```c
int n;
int* b;
cin>>n;
b = (int*) malloc(n*sizeof(int));
for(int i=0; i<n; i++)
    cin>>b[i];
```

In order to request space for `n` integers, we need to figure out how many bytes that is. That’s why we multiply with `sizeof(int)`. Using `sizeof(int)` is much better than hard-coding the constant 4, which may not be right on some hardware now or in the future.

Because `malloc` returns a `void*` (it does not know what we want to use the memory for), and we want to use it as an array of integers, we need to cast it to an `int*`.

For good coding practice, we should probably also check whether `b==NULL` before dereferencing it, but this example is supposed to remain short.

Another thing to observe here is that we can reference `b` just like an array, and we write `b[i]`. The compiler treats this exactly as `*(b+i)`, and, as you probably remember from the part about pointer arithmetic, this points to the `i`th entry of the array. In fact, that’s exactly how C/C++ internally treats all arrays anyway; basically, they are just pointers.

If we wanted to write `b[i]` in a complicated way by doing all the pointer arithmetic by hand, we could write instead `*((int*) ((void*) b + i*sizeof(int)))`. Obviously, this is not what we like to type (or have to understand), but if you understand everything that happens here, you are probably set with your knowledge of pointer arithmetic and casting.

To return the memory to the OS after we’re done using it, we use the function `free`, as follows:

```c
free(b);
b = NULL;
```

Note that `free` does nothing to the pointer `b` itself; it only deallocates the memory that `b` pointed to, telling the operating system that it is available for reuse. Thus, it is recommended that you immediately set the pointer to `NULL` so that your code does not attempt to tamper with invalid memory. If you reference `b` somewhere, you’ll just get a runtime error. If you don’t set `b=NULL`, the OS may give the memory to another variable, and you accidentally reference/overwrite that one. That kind of mistake can be much harder to detect, and is easily avoided by setting the pointer to `NULL` immediately after deallocating it.

4.2 C++ Style

C++ provides the `new()` and `delete()` functions that provide some syntactic sugar to C’s `malloc()` and `free()`. Basically, they relieve you from the calculations of the number of bytes needed, the casting of pointers, and provide a more “array-like” syntax. Our example now looks as follows:

```c++
int n;
int* b;
cin>>n;
b = new int[n];
```
Notice that there are no parentheses, but instead, we have brackets for the number of items. `new` figures out by itself how much memory is needed, and returns the correct type of pointer.

If we wanted space for just one integer, we could write `int *p = new int;` While this is not really very useful for a single integer, it will become very central to allocating objects later, where we often allocate one at a time dynamically.

To release memory, the equivalent of `free` is the `delete` operator, used as follows:

```c
delete [] b;
download p;
```

The first example deallocates an array, while the second deallocates a single instance of a variable (a single `int` in our example). This deallocates the memory pointed to by the pointers. As with `free`, it still leaves the pointers themselves pointing to the same memory location, so it is good style to write `b = NULL` or `p = NULL` after the `delete` commands.