Understanding how, when, where, and why memory is allocated is essential for anyone who aspires to be a good C++ programmer. In my experience, the single best tool for visualizing the allocation process is the **heap-stack diagram**, in which memory is divided into two regions. Dynamically allocated memory created using the `new` operator appears on the left hand side of the diagram, which represents the region of memory called the **heap**. Local variables declared as part of a procedure or function appear on the right side of the diagram, which represents the **stack**. The choice of left and right is in some sense arbitrary. Modern architectures, however, typically assign lower memory addresses to addresses in the heap than they do to addresses the stack, which supports the left-to-right layout.

For some reason that I’m not sure I as yet understand, students have not been as quick to pick up heap-stack diagrams as I had expected. Unlike the process of writing code, which invariably requires creativity, drawing heap-stack diagrams is essentially a mechanical activity. You should be able to follow the rules outlined in this handout and get full credit on any exam problem that asks you to create such a diagram. That fact, however, does not mean that the process is trivial or that the insights you get from making these diagrams are unimportant. When I’m helping students in the LaIR on the data-structure problems that come up later in the quarter, I’ve found that drawing these diagrams is the best way to help students get past those sticking points that make coding so frustrating. If you can spend a few minutes chugging this process and save hours of frustration as a result, that seems like a good tradeoff on the whole.

### Understanding the sizes of objects

Before going through the algorithm for generating heap-stack diagrams, I need to review the material from Wednesday’s lectures about the sizes of the different objects. Different data types require different amounts of memory. The primitive types, for example, can be sorted by size into the following categories:

<table>
<thead>
<tr>
<th>1 byte (8 bits)</th>
<th>2 bytes (16 bits)</th>
<th>4 bytes (32 bits)</th>
<th>8 bytes (64 bits)</th>
<th>16 bytes (128 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>short</td>
<td>int</td>
<td>long</td>
<td>long double</td>
</tr>
<tr>
<td>bool</td>
<td></td>
<td>float</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Even though this table of sizes is not too much to remember, we’ll make things even easier by stating up front that the only primitive types I will ever use in a heap-stack question are **char**, **bool**, **int**, and **double**, which means you can forget about the others.

In addition to the primitive types, you need to know the following rules for the sizes of the standard compound types:

- Every enumerated type defined using the `enum` keyword requires four bytes, making it the same as an `int`.
- Every structure type defined using the `struct` keyword must contain enough space to lay out the individual fields so that none of them overlap. We will not make any assumptions about the order or alignment of fields within a `struct`. If you want to simply lay them out in order one after another, that is always fine.
- Every array value requires the space necessary to store each of the elements. Thus, if an array has \(n\) elements and each element requires \(k\) bytes, the array as a whole will...
require \( n \times k \) bytes. Array elements, moreover, are always assigned consecutive locations in memory. The element at index position 0 appears at the beginning of the array, the element at index 1 appears immediately after that, and so on.

- Every pointer value is assumed to require four bytes, just as it would on any 32-bit architecture. In fact, given that the memory addresses I use range from 0000 to FFFF, I could have gotten away with 16 bits, but the larger size adds consistency to the diagrams.

An algorithm for generating heap-stack diagrams

The following process should be sufficient for creating any heap-stack diagram I ask you to produce. With practice, of course, you should be able to take various shortcuts and just write down the answer without going through all the steps.

1. **Start with an empty diagram.** Before you begin, draw a vertical line on the page to separate the heap space from the stack space. Both sides of the diagram are empty at the beginning. In a typical machine, the heap expands towards larger memory addresses and thus grows downward on the page; the stack, by contrast grows in the opposite direction and therefore grows upward on the page. My diagrams typically make 1000 the first address in the heap and FFFF the last byte in the stack, but these choices are merely conventions.

2. **Hand-simulate the program, allocating memory as you go.** The allocation of memory is a dynamic process that happens as the program runs. To figure out what memory looks like at a particular point, you need to trace through the program from the beginning. As you do so, the rest of the rules will apply at the appropriate time.

3. **Add a new stack frame for each function call.** Every time the program begins a function call (including the initial call to `main`), new memory is allocated on the stack side of the diagram to store what is called a stack frame. Drawing a stack frame is worth describing in a step-by-step process of its own.

   3a) **Start off by adding an “overhead” word represented as a rectangle filled in gray.** Inside the machine, the hardware always needs to track some information to keep track of the function-calling process. At a minimum, this information includes the address to which the function will return, but it usually also includes the contents of various internal registers that need to be saved across the function call. Since this information differs from machine to machine, there is no way to specify it exactly. In my own diagrams, I represent this information using a gray box that I draw as if it were a single word. When you draw your own diagrams, I won’t insist that you include this overhead word, but encourage you to do so, mostly because it has the effect of separating the stack frames visually.

   3b) **Include space in the stack frame for all local variables declared by the function.** The size of the stack frame you create depends on the number of variables it declares. Before you start your hand-simulation of the steps in the function, go through the code and find all of the local variable declarations that occur anywhere in the function. These declarations include the parameters to the function and any local variables declared in the body. Such declarations are usually easy to spot because they begin with a type name; the one place where they are easy to miss is in the declaration of loop indices in a `for` statement. For each variable you find, allocate as much space in the stack as that variable requires, and then label the space with the name of the parameter or local variable to which it corresponds. The only situation that requires any special consideration is when you have a parameter passed by reference, in which case the parameter occupies the space of a pointer rather than the actual value. Note that the order of the variables within a stack frame is arbitrary, which means that you can lay them out in whatever order you want.
3c) Initialize the parameter variables by copying the values of the actual arguments. After you have drawn the variables in the stack frame, you need to make sure that the parameters have the correct values. Remember that the association between actual arguments and parameter variables is determined by the order in which the values appear and never by name. Keep in mind also that arguments in C++ are passed by value unless the declaration of the parameter variable includes an & to indicate call by reference. This rule means that the values of the actual arguments are usually copied into the corresponding parameter variables. When a parameter uses call by reference, you don’t copy the value of the argument but instead assign its address to the pointer variable stored in the frame.

3d) Continue the hand simulation through the body of the function. Once you’ve set up the parameter values, you can then proceed with the hand-simulation of the statements in the function body. This will likely involve assignments (rule 4), memory allocations (rule 5), and nested function calls (recursive invocations of rule 3).

3e) Pop the entire stack frame when the function returns. When you finish executing a function, the stack frame that it was using is automatically reclaimed. On a diagram, you can simply cross out that space. The next function call will reuse the same memory.

4. Execute each assignment statement by copying the value of the expression on the right into the variable on the left. The nature of the copy depends to some extent on the type of value. If you assign a primitive value or an enumerated type, you simply copy the value. If you assign one struct to another, you copy every field. If you assign a pointer value to another, the pointer is copied, but not the underlying value. Moreover, because C++ treats array names as being synonymous with a pointer to their initial element, assigning an array name to a variable copies only the pointer and not the underlying elements. If you assign one object to another, the behavior depends on how that class defines assignment. I will talk about how object assignment works later in the quarter.

5. Allocate new heap memory when the program explicitly asks for it. The only time that a C++ program creates new memory in the heap is when the new operator appears explicitly in an expression. Whenever you see the keyword new, you need to draw space in the heap that is large enough to hold the value being allocated. For example, if you see the expression

```
new pointT
```

you need to reserve enough space in the heap to store a pointT, which requires eight bytes, four for each of the integer values it contains. Similarly, if you encounter the dynamic array allocation

```
new double[5]
```

you need to allocate 40 bytes in the heap, because each of the five elements requires the eight bytes necessary to store a double.

Once you have included the newly allocated memory on the heap side of the diagram, what you get back as the value of the new operator is a pointer to that block of memory. If you are assigning that value to a variable, you need to copy the pointer into the destination.

---

1 On Wednesday, I showed my programming age by using the old-style malloc form to indicate dynamic array allocation. In early versions of C++, that was how we did it. Sometime later, the designers of C++ extended the new operator so that it also works for arrays, as described in Chapter 2. The examples later in this handout correct Wednesday’s examples to use the more modern style.
A step-by-step example
The rest of this handout consists of a step-by-step analysis of the example I used in class on Wednesday, updated to use the modern syntax for dynamic array allocation. The updated program appears in Figure 1 at the bottom of the page.

Step 1. Start with an empty diagram.
The first step is to draw a line separating the page into a heap region and a stack region. In the interest of saving trees, I won’t draw this diagram until I put something on one side or the other.

Step 2. Hand-simulate the program, allocating memory as you go.
This step is really just an outline of the process to come. The first thing that happens is that the system invokes the main method, which brings us to . . .

Step 3. Add a new stack frame for each function call.
The main method has no parameters and three local variables: a pointT variable named pt, a double variable named total, and a dynamic integer array rather unimaginatively named array. The pointT variable requires 8 bytes (two fields times four bytes per field), the double variable requires 8 bytes, and the array variable (which is declared as a pointer) requires 4 bytes. This leads to the creation of the stack frame shown in the following diagram:

```
heap | stack
------|--------
array | FFE8
      | FFE4
      | FFC0
total | FFC8
      | FFC4
pt    | FFC0
      | FFC8
      | FFD4
```

Step 4. Execute each assignment statement by copying values.
From here, the next several steps continue the hand-simulation of the program through a series of assignment statements. Everything is straightforward until you get down to the new operator in the dynamic array initialization. Up to this point, all you do is fill in the

Figure 1. Heap-stack example

```c
int main() {  
  pointT pt;  
  double total = 0.0;  
  pt.x = 1;  
  pt.y = 2;  
  int *array = new int[5];  
  Nonsense(array, pt, total);  
  return 0;  
}

void Nonsense(int list[], pointT pt, double & sum) {  
  pointT *pptr = new pointT;  
  list[1] = pt.x;  
  sum += pt.y;  
}
```
values of the variables, which ends up generating a diagram that looks like this:

**Step 5. Allocate new heap memory when the program explicitly asks for it.**

The `new` operator in the main program just before the call to `Nonsense` is an explicit request for the allocation of heap memory. In this case, the code asks for space to hold an array of five integers, which means that a total of 20 bytes must be allocated in the heap. The address of that memory—which you get to choose arbitrarily—might be 1000. Assigning that value to the local variable `array` leads to the following situation:

**Step 3. Add a new stack frame for each function call.**

At this point, the program is set to call the `Nonsense` function. This time, there are parameters, and it is worth separating the process of reserving space in the frame from the process of initializing the parameters to the argument value. This time, there are four local variables: the parameters `list`, `pt`, and `sum`, and the local variable `pptr`. The sizes of these variables in the frame is calculated just as it was for `main` except for the fact that the `sum` parameter is passed by reference and therefore requires only enough space for the pointer. Before initialization, the diagram looks like this:
Step 3c. Initialize the parameter variables by copying the values of the actual arguments.

Copying the argument values into the local variables is reasonably straightforward. The only aspect of this step that might take a second look comes from the fact that `sum` is passed by reference, which means that you need to assign its address rather than its value:

![Diagram showing heap and stack memory allocation and usage](image)

Step 5. Allocate new heap memory when the program explicitly asks for it.

The first statement in the body of `Nonsense` is an invocation of the `new` operator, which allocates space for a `pointT` value. That value is allocated on the heap, and the address is then stored in the `pptr` variable. Rather than draw this intermediate step, it makes sense to initialize the values and show the final picture.

Step 4. Execute each assignment statement by copying values.

The remaining two assignment statements in `Nonsense` update the entry in `list[1]` and `sum`, which end up generating the following diagram at the maximum level of nesting:

![Diagram showing updated heap and stack memory](image)

Step 5. Pop the entire stack frame when the function returns.

The rest of the evolution of the program is returning from the two function calls, which involves popping each of the functions one at a time. The memory stored in the stack frames is automatically reclaimed; the memory allocated on the heap, by contrast, remains in place.