

Chapter 15

Dynamic Memory Allocation

Module 1: High-Level Programming 1

Copyright Notice

Copyright © 2017 DigiPen (USA) Corp. and its owners. All rights reserved.

No parts of this publication may be copied or distributed, transmitted, transcribed, stored in a retrieval system, or translated into any human or computer language without the express written permission of DigiPen (USA) Corp., 9931 Willows Road NE, Redmond, WA 98052

Trademarks

DigiPen® is a registered trademark of DigiPen (USA) Corp.

All other product names mentioned in this booklet are trademarks or registered trademarks of their respective companies and are hereby acknowledged.

Dynamic Memory Allocation in C and C++

Up until now, all memory allocation has been *static* or *automatic*

- The programmer (you) didn't have to worry about finding available memory; the compiler did it for you.
- You also didn't have to worry about releasing the memory when you were finished with it; it happened automatically.
- Static memory allocation is easy and effortless, but it has limitations.
- Dynamic memory allocation is under complete control of the programmer.
- This means that you will be responsible for allocating and de-allocating memory.
- Failing to understand how to manage the memory yourself will lead to programs that behave badly (e.g. run progressively slower, crash, etc.).

Comparing C and C++ memory allocating:

- In both C and C++, we can use *malloc* and *free*.

```
void *malloc(size_t size); // Allocate a block of memory
void free(void *pointer); // Deallocate a block of memory
```

To use *malloc* and *free* you need to include the following:

```
#include <stdlib.h> // malloc, free
```

The argument to *malloc* is the number of bytes to allocate:

Code	<pre>char *pc = malloc(10); // allocate memory for 10 chars int *pi = malloc(40); // allocate memory for 10 ints</pre>
Error	<pre>error C2440: 'initializing': cannot convert from 'void *' to 'char *' note: Conversion from 'void*' to pointer to non-'void' requires an explicit cast error C2440: 'initializing': cannot convert from 'void *' to 'int *' note: Conversion from 'void*' to pointer to non-'void' requires an explicit cast</pre>
Visualization	<p>The diagram shows two memory blocks. The first block, labeled 'pc', starts at address 100 and contains 10 question marks. The second block, labeled 'pi', starts at address 200 and also contains 10 question marks. Arrows point from the variable names 'pc' and 'pi' to the start of their respective memory blocks.</p>

Notice that there is no type information associated with *malloc*, so the return from *malloc* requires a cast to the correct type:

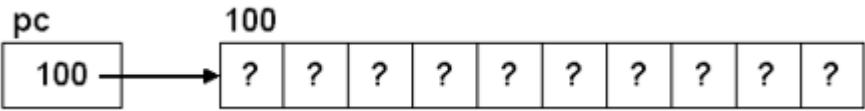
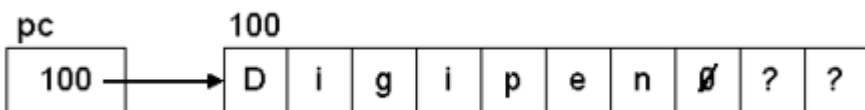
```
// Casting the return from malloc to the proper type
char *pc = static_cast<char *>(malloc(10)); // allocate memory for 10 chars
int *pi = static_cast<int *>(malloc(40)); // allocate memory for 10 ints
```

You should never hard-code the size of the data types, since they may change. Do this instead:

```
// Proper memory allocation for 10 chars
char *pc = static_cast<char *>(malloc(10 * sizeof(char)));

// Proper memory allocation for 10 ints
int *pi = static_cast<int *>(malloc(10 * sizeof(int)));
```

If the allocation fails **NULL** is returned, so you should check against `nullptr` after calling **malloc**:

Code	<pre>// Proper memory allocation for 10 chars char *pc = static_cast<char *>(malloc(10 * sizeof(char))); // If the memory allocation was successful if (pc != nullptr) { strcpy(pc, "Digipen"); // Copy some text into the memory std::cout << pc << std::endl; // Print out the text free(pc); // Release the memory pc = nullptr; // Safely point it at null } else { std::cout << "Memory allocation failed!\n"; }</pre>
Visualization	<p>After allocation</p>  <p>After strcpy</p> 

Notes:

- The memory allocated by `malloc` is uninitialized (random values).
- You need to initialize the memory yourself.
- If you want all of the memory to be set to zeros, you can use the `calloc` function instead:

```
// Allocates memory and sets all bytes to 0
void *calloc(size_t num, size_t size);
```

- Notice that `calloc` has two parameters:
 - 1) the number of elements
 - 2) the size of each element.

Code	<pre>// Allocate and initialize 10 chars to 0 char *pc = static_cast<char *>(calloc(10, sizeof(char)));</pre>
Visualization	<p>After calling <code>calloc</code>:</p>

- **`malloc`** and **`calloc`** are essentially the same, but, for obvious reasons, **`malloc`** is faster.
- If you are going to set the values of the memory yourself DO NOT use **`calloc`**. (It's an unnecessary waste of time.)

Accessing the allocated block:

Code	<pre>#include <iostream> #include <cstdlib> // malloc, free int main(void) { int SIZE = 10; int i, *pi; // allocate memory pi = static_cast<int *>(malloc(SIZE * sizeof(int))); // check for valid pointer (shorthand) if (!pi) { std::cout << "Failed to allocate memory." << std::endl; return -1; } // using pointer notation for (i = 0; i < SIZE; i++) { *(pi + i) = i; } // using subscripting for (i = 0; i < SIZE; i++) { pi[i] += i; } for (i = 0; i < SIZE; i++) { std::cout << *(pi + i) << " "; } // free memory free(pi); // safely, point at null pi = nullptr; return 0; }</pre>
------	---

Output	0 2 4 6 8 10 12 14 16 18
--------	--------------------------

By now it should be clear why we learned that pointers can be used to access array elements. With dynamic memory allocation, there are no *named* arrays, just pointers to contiguous (array-like) memory and pointers *must* be used.

Dynamic Memory Allocation in C++

- In C++, we have a better alternative to `malloc` and `free`.
- In C++, we can use `new` to allocate memory, and `delete` to free the memory.
- There are also array versions of `new` and `delete` (for allocating arrays).
 - `new` and `delete` are keywords.
- Both methods return pointers, making the use of that memory is identical regardless of allocation method.

Example 1: Simple allocation of built-in type:

```
// Dynamically allocate space for an int
int *i1 = static_cast<int *>(malloc(sizeof(int))); // C and C++ (4 bytes)
int *i2 = new int;                               // C++ only (4 bytes)

// Use i1 and i2

// Release the memory (programmer)
free(i1); // C and C++
delete i2; // C++ only

i1 = NULL; // Safely reassign to NULL, C and C++
i2 = nullptr; // Safely reassign to nullptr, C++ only
```

Note:

- `malloc` is used in both C and C++, but `static_cast` is purely C++
- In pure C, we use the C-style cast:

```
int *i1 = (int *)malloc(sizeof(int));
```

We will continue using examples with `static_cast`, as it is the preferred method of casting in C++.

More Notes:

- Every variable is allocated either statically or dynamically.
- Dynamic allocation is more expensive
 - Memory from the heap is slower than the stack
 - You have to be extra cautious to safeguard against memory leaks (e.g. check against `nullptr`, delete as needed, avoid double deletion, etc.)
- Overall, prefer static over dynamic allocation; it is safer and faster.
- Dynamic allocation is for those things that have the ability to change dramatically, such that a static version would incur too much wasted space or would unnecessarily limit the output of a program *due to its set size*.
- `free` and `delete` are NOT interchangeable. Ensure you match the correct deallocation function based on how you allocated it. (If you had the compiler statically allocate it, don't clean it up!)

Example 2: Allocating arrays of built-in types:

```
// Allocate space for array of 10 chars and 10 ints (C and C++)
char *p1 = static_cast<char *>(malloc(10 * sizeof(char))); // 10 bytes
int *p2 = static_cast<int *>(malloc(10 * sizeof(int))); // 40 bytes

// Allocate space for array of 10 chars and 10 ints (C++ only)
char *p3 = new char[10]; // 10 bytes
int *p4 = new int[10]; // 40 bytes

// Use p1, p2, p3, p4 ...

// Release the memory (programmer)
free(p1); // C and C++
free(p2); // C and C++
delete[] p3; // C++ only (array delete)
delete[] p4; // C++ only (array delete)

p1 = p3 = nullptr; // safely set the pointers
p2 = p4 = nullptr; // to nullptr, C++ only
```

Note:

- `delete` and `delete []` are NOT interchangeable. Ensure you match the correct deallocation function based on how you allocated it.
Using the non-array version on the array will leave nearly all the elements still allocated, while the opposite may attempt to deallocate things that have no business being deallocated.
- Deallocating something that shouldn't be (e.g. double deleting a pointer, mismatching `new/delete[]`, etc.) renders your program **undefined**.
- It is safe to `free/delete/delete []` a pointer with a value of `nullptr`. These functions check internally and will result in a no-operation in the case of a `nullptr` value. (This same benefit is gained using `NULL`, or 0, in C.)
This means setting the pointers to `nullptr` after deallocation, greatly reduces the chance of double deletion.
- Setting things to `nullptr` also helps you identify paths of code that may be trying to use deallocated or uninitialized data via that pointer.

Example 3: Allocation of a struct:

```
// On-screen graphic
struct Sprite
{
    double x, y;
    int weight;
    int level;
    char name[20];
};

void foo(void)
{
    Sprite s1; // Allocated on the stack (handled by compiler)

    // Dynamically allocate on the heap (handled by the programmer)
    // 44 bytes (C and C++)
    Sprite *s2 = static_cast<Sprite *>(malloc(sizeof(Sprite)));

    // 44 bytes (C++ only)
    Sprite *s3 = new Sprite;

    s1.level = 1; // s1 is a Sprite struct
    s2->level = 2; // s2 is a pointer to a Sprite struct
    s3->level = 3; // s3 is a pointer to a Sprite struct

    // Other stuff ...

    // Release the memory (programmer)
    free(s2); // C and C++
    delete s3; // C++ only

    s2 = s3 = nullptr; // safely set to nullptr, C++ only
} // s1 goes out of scope and the memory is released automatically
```


Example 4: Allocating arrays of structs (user-defined type):

```

void bar(void)
{
    // Allocated array of 10 Sprites (handled by compiler)
    Sprite s1[10];

    // Dynamically allocate array of 10 Sprites (handled by the programmer)
    // 440 bytes (C & C++)
    Sprite *s2 = static_cast<Sprite *>(malloc(10 * sizeof(Sprite)));
    // 440 bytes (C++ only)
    Sprite *s3 = new Sprite[10];

    s1[0].level = 1; // s1[0] is a Sprite struct
    s2[0].level = 2; // s2[0] is a Sprite struct
    s3[0].level = 3; // s3[0] is a Sprite struct

    s2->level = 4; // Does this work?
    s3->level = 5; // Does this work?

    // Release the memory (programmer)
    free(s2);      // C and C++
    delete[] s3;  // C++ only (array delete)

    s2 = s3 = nullptr; // safely set to nullptr, C++ only
} // s1 goes out of scope and the memory is released automatically

```

Since `p->m` is equivalent to `(*p).m`

```

s2->level = 4; // => (*s2).level = 4;
s3->level = 5; // => (*s3).level = 5;

```

When `s2` and `s3` are dereferenced they are pointing at the beginning of their respective arrays, which shares the same address as the first element in those arrays.

```

*s2 is the same as s2[0]
*s3 is the same as s3[0]

```

Thus, using the member-access operator, allows those lines of code to set the level member of the first elements.

e.g. Equivalent lines of code:

```

s2[0].level = 4; // s2[0] is a Sprite struct
s3[0].level = 5; // s3[0] is a Sprite struct

```

The answer is yes, those lines of code *do* work.

Example 5: Modified Sprite struct:

```
// On-screen graphic, new version
struct Sprite2
{
    double x, y;
    int weight;
    int level;
    char *name; // not an array
};

void baz(void)
{
    // Dynamically allocate Sprite2 on the heap (handled by the programmer)
    // 28 bytes (C and C++)
    Sprite2 *s1 = static_cast<Sprite2 *>(malloc(sizeof(Sprite2)));
    // 28 bytes (C++ only)
    Sprite2 *s2 = new Sprite2;
    Sprite2 *s3 = new Sprite2;

    // Statically allocate char array
    char str[6] = "Stack";

    // Assign values to name member
    // Dynamically allocate 10 chars on the heap (programmer)
    // 10 bytes (C and C++)
    s1->name = static_cast<char *>(malloc(10 * sizeof(char)));
    // 10 bytes (C++ only)
    s2->name = new char[10];
    // Point at an already existing statically allocated value
    s3->name = str;

    // Release memory for chars
    free(s1->name); // C and C++
    delete[] s2->name; // C++ only (array delete)
    // Don't release s3->name, since it DID NOT allocate memory
    // safely set to nullptr, C++ only
    s1->name = s2->name = s3->name = nullptr;

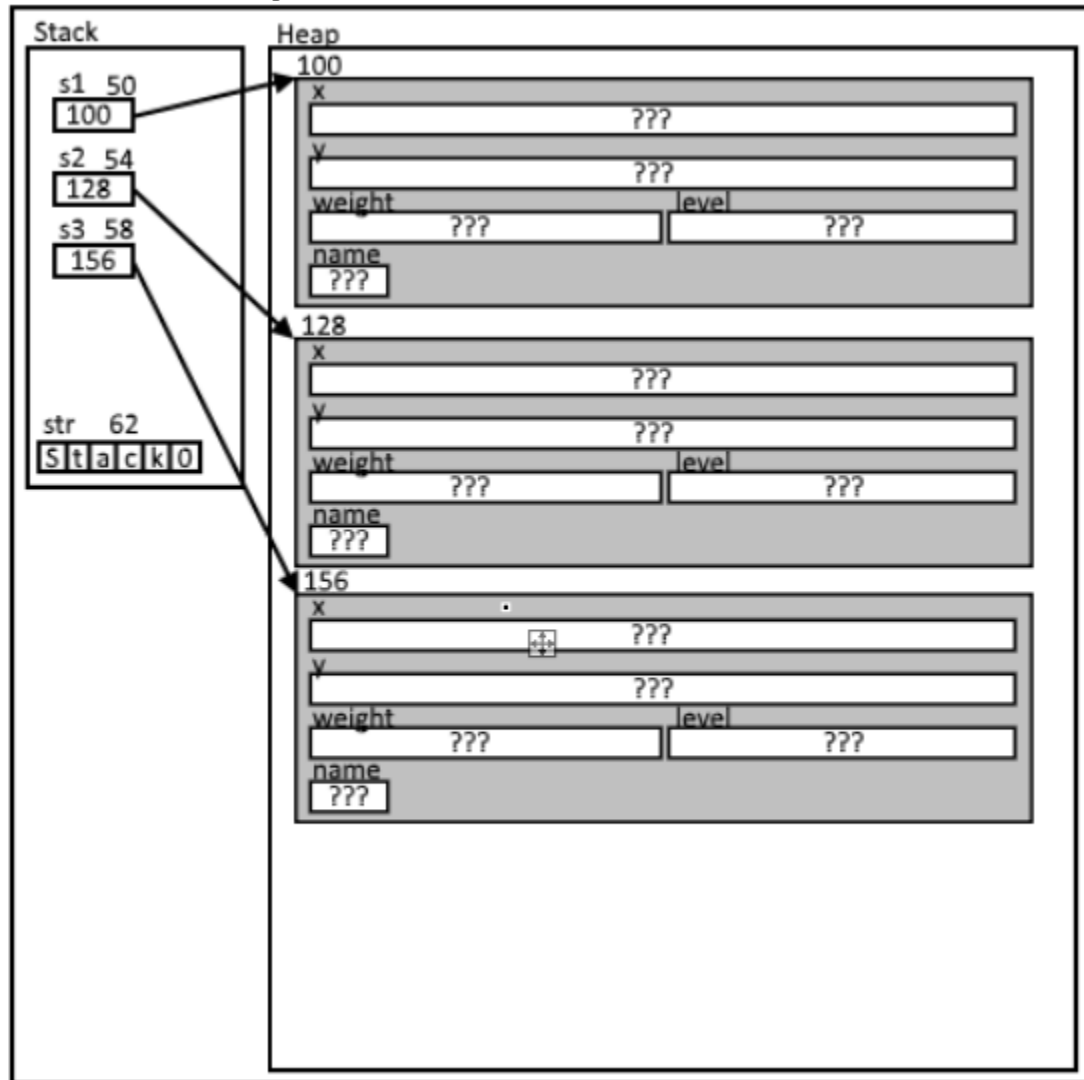
    // Release memory for Sprite2
    free(s1); // C and C++
    delete s2; // C++ only
    delete s3; // C++ only

    // safely set to nullptr, C++ only
    s1 = s2 = s3 = nullptr;
}
```

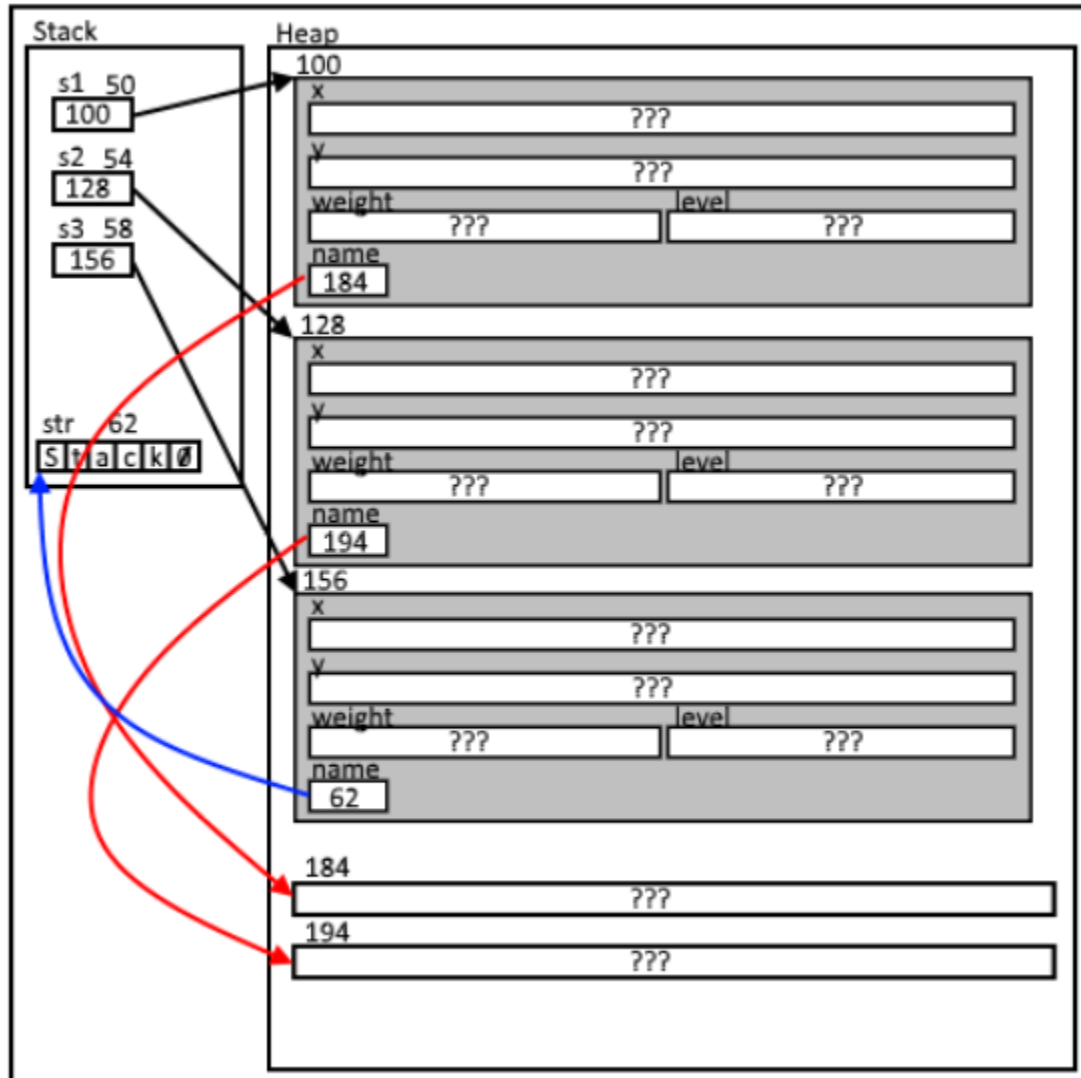
Note:

- When you have dynamically allocated a pointer, you may also need to allocate the memory it points at, too. You may, instead, choose to use the pointer to point at statically allocated items (as in the case of `s3`), this depends on the design and usage of your code.

After allocating s1, s2, s3 and str:



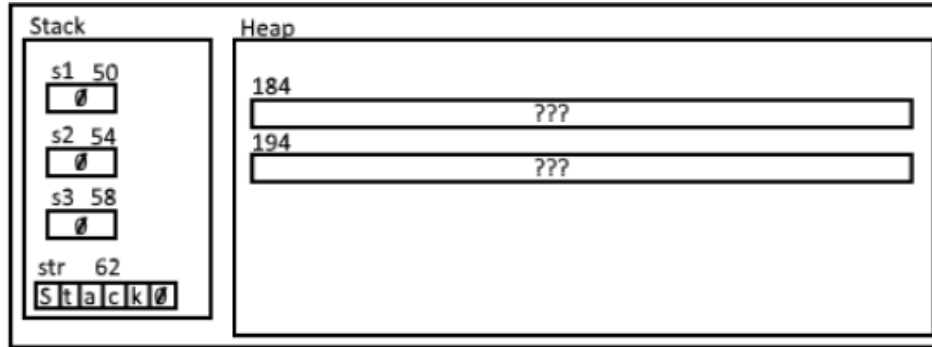
After setting `s1->name`, `s2->name` and `s3->name`:



The red arrows highlight the pointers pointing at other items allocated on the heap, while the blue arrow corresponds to the one pointing to an item on the stack.

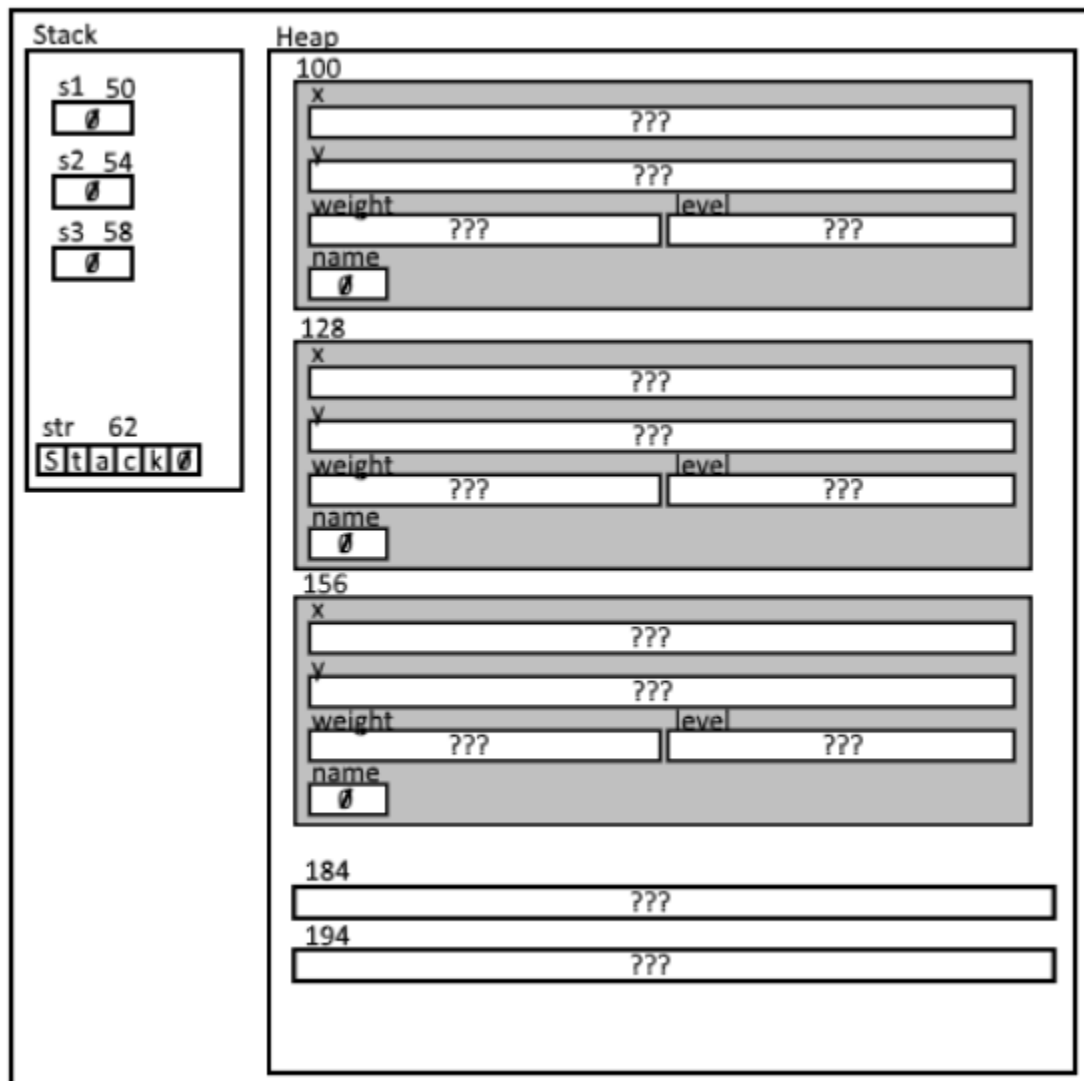
- When you have allocated objects that include pointers that point at dynamically allocated memory you must deallocate those pointers before the containing object (be it a struct, array, etc.). e.g. `s1->name` and `s2->name` MUST be deallocated before `s1` and `s2` respectively, otherwise you won't have access to the address to pass it to its deallocation function.

Visual of deleting out of order:



Notice that no variables have access to either the address 184 or 194 to clean up the memory that was left on the heap after deleting s1, s2 and s3.

- Remember, setting a pointer to `NULL`, `0`, or `nullptr` is not a deallocation! Pointing the pointers at nothing doesn't clean up the heap:



You MUST call a deallocation function (e.g. `free`, `delete`, or `delete[]`) to clean up the memory and avoid leaks.

Example 6: Stack and heap allocations:

```
void gar(void)
{
    // Allocate 10 characters on the stack
    char a[10];

    // Point at the first element (C and C++)
    char *p1 = a;

    // Allocate space for array of 10 chars and 10 ints (C and C++)
    char *p2 = static_cast<char *>(malloc(10 * sizeof(char))); // 10 bytes
    // Allocate space for array of 10 chars and 10 ints (C++ only)
    char *p3 = new char[10]; // 10 bytes

    // All three pointers work in the exact same way
    // There is no way to tell how the memory was allocated
    // Therefore the programmer must think ahead about how the
    // memory will be cleaned up, to ensure it is both done and
    // done correctly.

    // Release the memory
    free(p2); // C and C++
    delete[] p3; // C++ only

    p2 = p3 = nullptr; // safely set to nullptr, C++ only
} // a is automatically released here.
//calling free or delete on a is very dangerous!
```

- One of the biggest culprits of memory leaks is missing a path of code that may bypass the clean-up code.

Example 7: Branching code & memory leaks:**Code**

```

#include <iostream> // std::cout, std::cin
#include <cstring> // strcpy, strcat, strlen
#include <cstdlib> // srand, rand
#include <ctime> // time

int main(void)
{
    const int iceCreamCake = 1;
    const int cheesecake = 2;
    int choice;

    std::cout << "Press 1 for ice cream cake or 2 for cheesecake: ";
    std::cin >> choice;
    std::cout << std::endl;

    srand(time(0));

    char *msg = nullptr;
    char *str = "You ordered "; // Easy to change later

    // dynamically construct message based
    // on which option was chosen
    switch (choice)
    {
    case iceCreamCake:
        msg = new char[strlen("ice cream cake.") + strlen(str) + 1];
        if (msg) // if successfully allocated
        {
            strcpy(msg, str); // create the
            strcat(msg, "ice cream cake."); // message
            std::cout << msg; // print it out
            delete[] msg; // clean up the memory
            msg = nullptr; // safely set to nullptr
        }
        break;

    case cheesecake:
        msg = new char[strlen("cheesecake.") + strlen(str) + 1];
        if (msg) // if successfully allocated
        {
            strcpy(msg, str); // create the
            strcat(msg, "cheesecake."); // message

            //----- add some new code to print something -----
            // else every once in a while
            if (rand() % 2)
            {
                // Prints out the statement
                std::cout << "Cheesecake is currently out of stock.";
                // don't want to print other msg, so jump out...
                break;
            }
            //-----

            std::cout << msg;
            delete[] msg; // Sometimes this doesn't get run!
            msg = nullptr; // This means it sometimes leaks!
        }
        break;
    }
}

```

	<pre> default: std::cout << "You seem to prefer neither option."; } return 0; } </pre>
Explanation	<p>When there was one path through this case statement the memory was correctly accounted for. The added code path through this case (made by branching at the if condition and jumping out with <code>break</code>) created a leak by never hitting the clean-up code that follows.</p> <p>This is a very contrived example, but it illustrates how simple, poorly placed additions to code can create leaks due to branching code paths.</p>

Dynamically Allocating a 2D Arrays

The first instinct may just be to just make a big array. E.g. If I want 12 doubles:

```

const int ROWS = 3;
const int COLS = 4;

double *pd = new double[COLS * ROWS];

```

From there, use the subscript operator and call it a day.

When you reexamine the definition, you'll quickly see this does not work:

`pd[1][3]` is equivalent to `*(pd+1)[3]` and equivalent to `*(*(pd+1)+3)`

The problem is that `*(pd+1)` is the dereferencing a pointer to a double, thus revealing the double underneath.

Adding to the value of the double gives you simple addition -*NOT the intended pointer arithmetic that subscripting aims to achieve!*

Moreover, dereferencing a non-pointer type (`double`) doesn't work.

e.g. Looking at the types evolve as the expression resolves:

```

given: *(*(pd+1)+3), where pd is a double*
then:  *(*(pd+1)+3) => *(*(double* + 1) +3) // pointer arithmetic
                        => *(*(double*) +3)  // dereference
                        => *((double) +3)     // addition
                        => *(double)          // ILLEGAL
                        => trying to dereference a non-pointer type, error

```


In order to access these via a single pointer you have to do much of the math yourself:

```
int row = 1, column = 3;
double value;
// Access via single pointer using pointer arithmetic
// and/or subscripting. These statements are all equivalent.
value = pd[row * COLS + column];
value = *(pd + row * COLS + column);
value = (pd + row * COLS)[column];
```

Alternately, to enable the use of multiple subscripts, we can set up the allocation to include more pointers, so the pointer arithmetic and dereferencing done by the subscripts works.

Using these definitions:

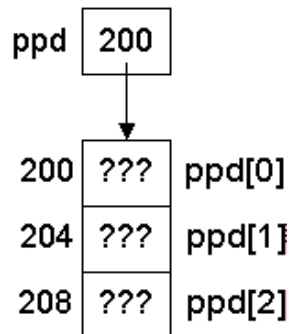
```
const int ROWS = 3;
const int COLS = 4;
```

Create a variable that is a *pointer to a pointer to a double*

```
double **ppd;
```

Allocate an array of 3 (ROWS) pointers to doubles and point *ppd* at it:

```
ppd = new double *[ROWS];
```



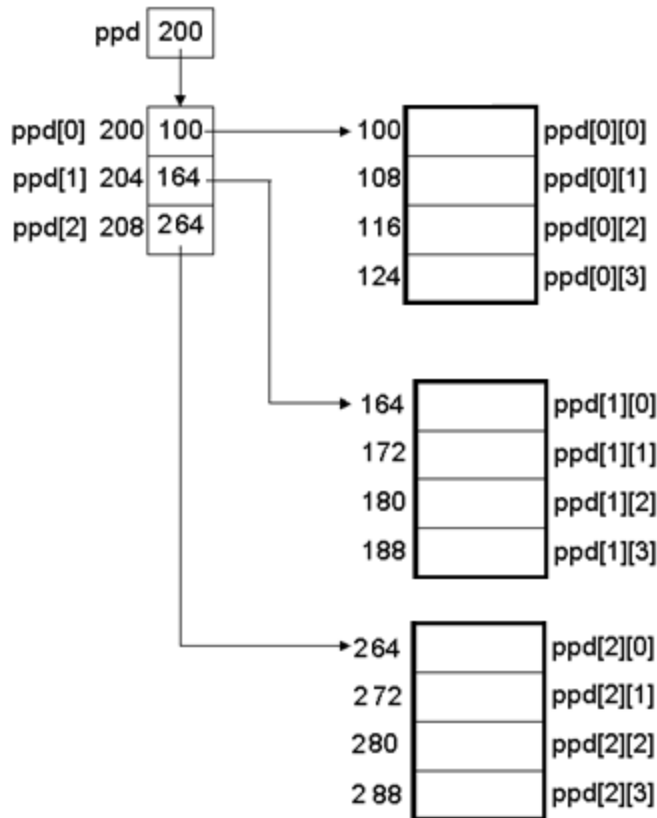
In each element of *ppd*, allocate an array of 4 (COLS) pointers to doubles:

```
ppd[0] = new double[COLS];
ppd[1] = new double[COLS];
ppd[2] = new double[COLS];
```

Of course, for a large array, or an array whose size is not known at compile time, you would want to set these in a loop:

```
for (int r = 0; r < ROWS; ++r)
{
    ppd[r] = new double[COLS];
}
```

This yields the diagram:



If we look at the subscript operator unfold, now that we have changed the type we can see it resolves correctly:

e.g. Looking at the types evolve as the expression resolves:

knowing `ppd[1][3]` is equivalent to `*(*(ppd+1)+3)`

given: `*(*(ppd+1)+3)`, where `ppd` is a `double**`

```
then: *(*(ppd+1)+3) => *(* (double** + 1) +3) // pointer arithmetic
      => *(* (double**) +3) // dereference
      => *((double*) +3) // pointer arithmetic
      => *(double*) // dereference
      => double // correct value!
```

Full code:

```
const int ROWS = 3;
const int COLS = 4;

// Creating the 2D array
double **ppd = new double *[ROWS];
for (int r = 0; r < ROWS; ++r)
{
    ppd[r] = new double[COLS];
}

// Fill the 2D array with zeros
for (int r = 0; r < ROWS; ++r)
{
    for (int c = 0; c < COLS; ++c)
    {
        ppd[r][c] = 0.0;
    }
}

// Use the array...

// Deleting the 2D array
// delete in the opposite order of creation
for (int r = 0; r < ROWS; ++r)
{
    delete[] ppd[r]; // delete each allocated row with array delete
    ppd[r] = nullptr; // safely set to nullptr
}

// Use array delete, since array new was used in allocation
delete[] ppd;
ppd = nullptr; // safely set to nullptr.
```