# Chapter 8

# Dynamic Memory Allocation

CS185

# 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 (crash).

Comparing C and C++ memory allocating:

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

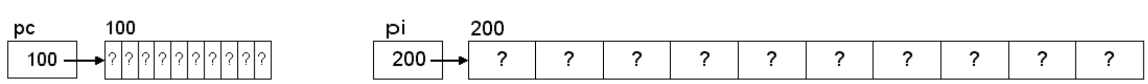
```
void *malloc( size_t size ); /* Allocate a block of memory   */
void free( void *pointer );  /* Deallocate a block of memory */
```

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:

```
char *pc = malloc(10); /* allocate memory for 10 chars */
int *pi = malloc(40);  /* allocate memory for 10 ints  */
```




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

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

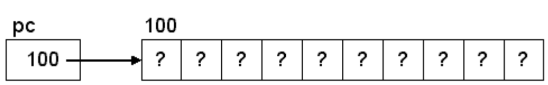
You should never hard-code the size of the data types, since they may change. Do this instead:

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

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

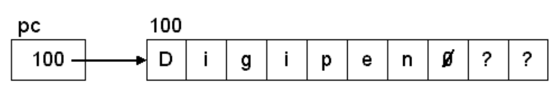
If the allocation fails, **NULL** is returned so you should check the pointer after calling ***malloc***.

```
  /* Allocate some memory for a string */
char *pc = (char *) malloc(10 * sizeof(char));

  /* If the memory allocation was successful */
if (pc != NULL)
{
  strcpy(pc, "Digipen"); /* Copy some text into the memory */
  printf("%s\n", pc);    /* Print out the text             */
  free(pc);              /* Release the memory             */
}
else
{
  printf("Memory allocation failed!\n");
}
```

**After allocation**




**After strcpy**




**Notes:**

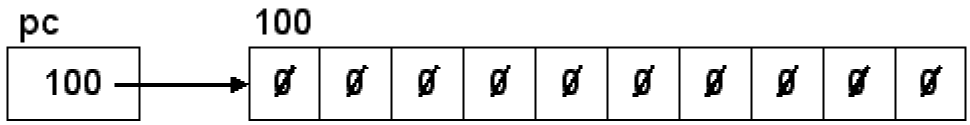
- The memory allocated by `malloc` is uninitialized (random values).
- You need to initialize the memory yourself.
- If you want all of the memory to be set to zeros, you can use the `calloc` function instead:

```
/* 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.

```
  /* Allocate and initialize 10 chars to 0 */
char *pc = (char *) calloc(10, sizeof(char));
```

- After calling `calloc`:




- ***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:

```
void test_malloc(void)
{
	int SIZE = 10;
	int i, *pi;

	/* allocate memory */
	pi = (int *)malloc(SIZE * sizeof(int));

	/* check for valid pointer */
	if (!pi)
	{
		printf("Failed to allocate memory.\n");
		return;
	}

	/* 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++)
	{
		printf("%i \n", *(pi + i));
	}

	/* free memory */
	free(pi);
}
```

**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.
- Although the way of allocating memory has changed with C++, the use of that memory is identical to C.

## Example 1: Simple allocation of built-in type:

```
void f1(void)
{
       // Dynamically allocate space for an int
       int *i1 = (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
}
```

## Example 2: Allocating arrays of built-in types:

```
void f2(void)
{
       // Allocate space for array of 10 chars and 10 ints (C and C++)
       char *p1 = (char *)malloc(10 * sizeof(char)); // 10 bytes
       int *p2 = (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)
}
```

**Example 3: Allocation of a struct:**

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

void f3(void)
{
	Sprite s1; // Allocated on the stack (handled by compiler)

	// Dynamically allocate on the heap (handled by the programmer)
	Sprite *s2 = (Sprite *)malloc(sizeof(Sprite)); // 44 bytes (C and C++)
	Sprite *s3 = new Sprite;                       // 44 bytes (C++ only)

	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

} // s1 goes out of scope and the memory is released automatically
```

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

```
void f4(void)
{
	// Allocated array of 10 Sprites (handled by compiler)
	Sprite s1[10];

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

	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)

} // s1 goes out of scope and the memory is released automatically
```

### 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 f5(void)
{
       // Dynamically allocate Sprite2 on the heap (handled by the programmer)
       Sprite2 *s1 = (Sprite2 *)malloc(sizeof(Sprite2)); // 28 bytes (C and C++)
       Sprite2 *s2 = new Sprite2;                         // 28 bytes (C++ only)

       // Dynamically allocate 10 chars on the heap (programmer)
       s1->name = (char *)malloc(10 * sizeof(char)); // 10 bytes (C and C++)
       s2->name = new char[10];                      // 10 bytes (C only)

       // Release memory for chars
       free(s1->name);     // C and C++
       delete[] s2->name; // C++ only (array delete)

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

### Example 6: Stack and heap allocations:

```
void f6(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 = (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

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

} // a is automatically released here.
  //calling free or delete on a is very dangerous!
```

# Dynamically Allocating a 2D Arrays

Using these definitions:

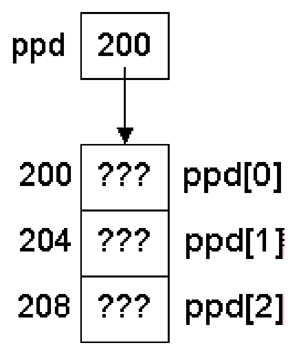
```
#define ROWS 3
#define COLS 4
```

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:

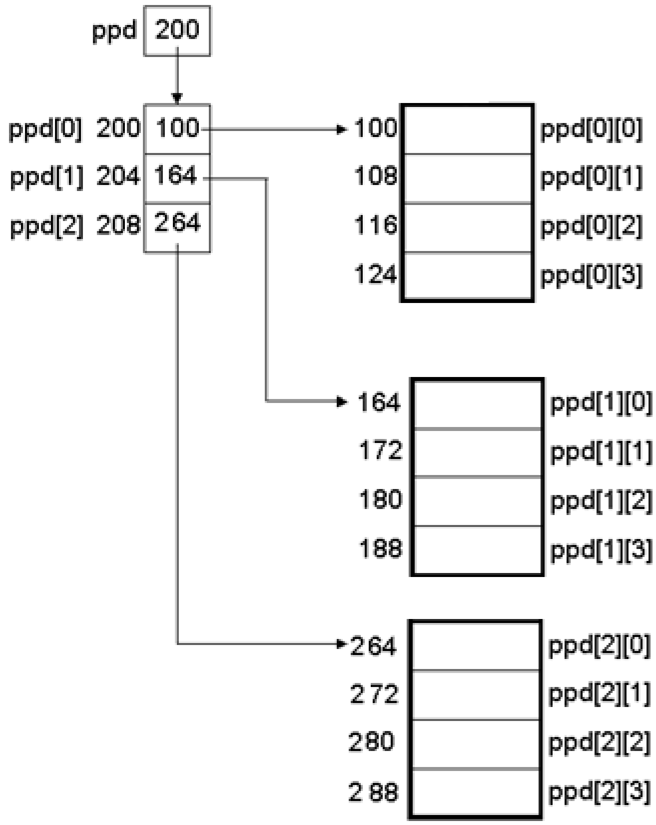
```
ppd[0] = new double [COLS];
ppd[1] = new double [COLS];
ppd[2] = new double [COLS];
```

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:




**Full code:**

```
#define ROWS 3
#define 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;
	}
}
```