Memory & storage

How to use dynamic memory

Dynamic memory is a term given to a concept which allows programmers to create and destroy persistent storage space at runtime. One of the major differences separating dynamic memory allocations from global variables is the life-time of the data.

A global variable, being anything that is declared outside of a function, has its storage size calculated at compile time and is created before your sketch runs the setup() function, or more accuratly, before C++ runs main(). Its life-time spans the entire application, as in, it is not destroyed and consequently its memory is not available for use anywhere else.[1]

The same consequences apply also to local variables marked static and static data members of structs, classes and unions as they too are stored in the heap. Static globals have the same life-time as regular globals, the static keyword affects its visibility to other parts of the program.

The life-time of dynamically created variables is completely up to your code. Which means you can use dynamic allocations to manage large working spaces for your application, while being able to free up the memory for other large tasks to run. Of course the flexibility comes at a price of complexity, as you are resposible for cleaning up memory when finished and failure to do this can result in extemely hard bugs to trace, especially as the Arduino has limited debugging capabilities.

As the Arduino system uses GCC and its C++ compiler, you have access to both C and C++ memory allocation methods. Having a mild understanding of pointers will make this article far easier to understand.

- **C++ style allocations**
- **C style allocations**

For an insight into pointers the people at *cplusplus.com* have written a nice article here: http://www.cplusplus.com/doc/tutorial/pointers/.

C++ style allocations

Although my opinion is biased towards C++, I recommend using these methods as they fully encapsulate the behaviour of C style allocations, while providing much more functionality if needed.

- **Dynamic variables/objects.**

  Using new to dynamically create a variable is quite simple. There are a number of ways to create a variable, which method depends on your application and the requirements set by the variables type.

  For primitive types, the usage of new is straight forward with three methods to create your variable.

```
//-1. Create an uninitialized variable:
float *myPtr = new float;
```

```
//-2. Create a default initialized variable:
float *myPtr = new float();

//-3. Create an initialized variable:
float *myPtr = new float( 1.23f );
```

When creating a POD type, there are a few more constraints. The third version cannot be used to initialize the objects members directly, however you can pass in a fully formed object to be copied, or alternatively an anonymous temporary to replicate default initialization.

```
struct POD{
  int value;
};

struct OBJ{
  OBJ() : value() {}
  OBJ( int v ) : value( v ) {}
  int value;
};

POD *myPtr = new POD; //Uninitialized.
OBJ *myPtr = new OBJ; //Default initialized.

POD *myPtr = new POD(); //Default initialized.
OBJ *myPtr = new OBJ(); //Default initialized.

//POD *myPtr = new POD( 5 ); //Error.
OBJ *myPtr = new OBJ( 5 ); //Initialized.

//Default initialize.
POD *myPtr = new POD( POD() );

//Invoke the copy constructor ( compiler generated ).
POD p = { 5 }; //Auto variable.
POD *myPtr = new POD( p ); //p copied to dynamic variable.
```

All vairbales created with new should[2] be destroyed once they are no longer in use. You can achieve this with one simple statement: delete myPtr;

If you plan to reuse the pointer variable it is a good idea to set it to zero while not in use. A standard implementation of delete will do nothing on a null pointer[3] which will prevent errors if the pointer is accidentially deleted twice.

- **Dynamic arrays.**

Similar to dynamic variables, new can also allocate arrays of variables. The syntax used is not much different, however it is a little more restrictive with regards to initializing the arrays elements. An arrays length is specified using square brackets and when deleting an array, you must always remember the empty square brackets to signify an array type.

```
//Create a dynamic array of ints with 16 elements.
int *myArr = new int[ 16 ];

//Use myArr just like a normal array.
myArr[ 0 ] = 4;

//Clean up once finished with the array.
delete [] myArr;
```

The example above simply allocates an array of uninitialized data. Unlike static arrays, you cannot initialise the array elements to a specific value, however you can default initialize it which can be useful.

```
//Default initialise the array:
int *myArr = new int[ 16 ]();
```

- **Allocating a block of memory.**

A very basic operation you can do with the new operator is to allocate a raw chunk of memory. This feature is the C++ way of implementing C's malloc() function, and in the Arduino environment it is equivalent as new calls malloc() internally.

```
//Calculate the length of 10 float objects.
int len = sizeof( float ) * 10;

//Allocate 'len' bytes of data.
void *ptr = ::operator new( len );
```

The above code successfully allocates enough space for ten floats, however the result is stored in a void* which is not entirely useful in this situation, but is the type returned by new which contians the location/address of the first byte. Void pointers do not contain any type information and therefore are not usable as an array placeholder like pointers of a real type. Accessing the data at even the byte level requires at least a char* type.

To access the raw data in the allocation we use what is called a cast. A cast is what C++ uses to convert between types and is denoted by a type name in brackets. Casts are outside of this topic and will be covered in a future FAQ, only a small example is explained here. In the snippet below a cast is used to represent a void* as a float* to then assign to a float* object.

```
//Cast 'ptr' to a float pointer.
float *floatArray = ( float* ) ptr;

//Or simply cast result of 'new'
float *floatArray = ( float* ) ::operator new( len );

//The pointer to the allocation can be used like an array.
floatArray[ 0 ] = 1.23f;
floatArray[ 1 ] = 4.56f;
```

You may have noticed this method of allocation uses a slightly different syntax to utilize new. When typically using the new keyword in your code, the new function is implicitly called and it returns a pointer to an allocation. This allocation is used as the location to construct your object. As this method of using new does not construct an object you cannot use the new keyword and have to explicitly call the new function.

C style allocations

- **Allocating a block of memory**

  Standard C includes a few functions which provide basic memory management. Unlike the C++ version, these functions do not allow initialization of objects, but work with raw memory allocations. The main allocation tool is a function called malloc(), it takes a single parameter of the size to allocate and like C++ new, it returns a void* type. Its lifetime is also under your control and free() is needed to be called to deallocate the memory for use elsewhere.

```
//Allocate an uninitialized block of memory big enough to hold 16
 ints.
int *myArr = ( int* ) malloc( sizeof( int ) * 16 );

//Use myArr just like a normal array.
myArr[ 0 ] = 4;

//Clean up once finished with the allocation.
free( myArr );
```

Even though C style allocations do not allow initialization of objects, you can mimmic the behaviour of default zero initialization using a function called calloc(). This function returns the allocation with all bytes equal to zero, and allows a slightly different syntax over malloc().

```
//Allocate a zero initialized block of memory big enough to hold
16 ints.
int *myArr = ( int* ) calloc( 16, sizeof( int ) );

//Use myArr just like a normal array.
myArr[ 0 ] = 4;

//Clean up once finished with the allocation.
free( myArr );
```

A feature which C++ doesn't support directly is resizing allocations. Using the function realloc(), you can grow or shrink your allocation accordingly. This function can be quite useful as it will maintain the data already in the allocation which will reside in the new sized allocation. If the allocation is reduced in size, the elements that do not get copied are lost.

```
//Allocate 16 integers.
int *myArr = ( int* ) calloc( 16, sizeof( int ) );

//Resize to twenty values.
myArr = ( void* ) realloc( myArr, 20 * sizeof( int ) );

//Clean up once finished with the allocation.
free( myArr );
```

One final point to note, is the above code initially allocates its memory using calloc(), however the 4 new values added by realloc() are uninitialized and should be zeroed if you read from them before writing.

Tag Notes:

1. When a standard C++ application exits, the global variables have a chance to shutdown and run their destructors. In an Arduino environment the system runs an infinite loop so as to never exit until the power is cut to the board.
2. On embedded systems if you allocate dynamic memory that is intended to exist the entire life-time of the application, there is no need to delete it, and could waste flash space storing unused instructions. It is good practice however to satisfy the new/delete combination as not deleting your code on a multi-process machine could render portions of memory unusable even after your application exits.
3. The Arduino core implementation does not satisfy the standard as the delete definition skips the check for a null pointer, however as it uses free(), it will not cause undefined behaviour as a null pointer is ignored. You can see a full set of the standards satisfying drop in replacements from this FAQ: here.

Memory & storage

This FAQ is part of a series, the other articles can be accessed here.

- **Can I use new & delete with Arduino**
- **Mixing malloc and new**
- **Using placement new & delete**

Unique solution ID: #1025
Author: Christopher Andrews
Last update: 2014-04-10 17:23