Morning Musings

I'm not ready to wake up yet...

Placement New, Memory Dumps, and Alignment

Aug 23rd, 2015 | Comments

Table of Contents

This is an indepth post about advanced memory topics in C++ such as placement new and alignment. The topics covered here will be used in a future post to create a memory pool.

Memory

Generally, in C++ there are three places you can store your data:

1. On the stack (local variables)
2. On the heap (new / delete)
3. In the static data section (static variables)

Normally, when using the heap, you would use the following command:

```
Foo* foo_ptr = new Foo();
```

This would invoke the following actions behind the scenes:

1. Make an operating system call to allocate a chunk of memory of size `sizeof(Foo)`
2. Manage that memory with the heap
3. Call the constructor of Foo to build an object at that memory location
4. Initialize `foo_ptr` with the address of the object

Later on, you would call:

```
delete foo_ptr;
```

This would:

1. Call the destructor of Foo
2. Have the heap give the memory back to the operating system

This is how `new` and `delete` work, and most C++ programmers should be familiar with them.

> **New and Malloc**
> You should never `delete` memory allocated with `malloc`, and you should never `free` memory allocated with `new`. `malloc` and `free` do raw memory allocations, while `new` and `delete` are also responsible for calling constructors and destructors.

Placement New

C++ offers a different "flavor" of `new` called "placement new". Placement new gives the user finer control about where the object gets constructed by allowing the object to be "placed" at a specified memory address; in other words, the heap allocation step is bypassed.

Placement new has a few specialized uses. One is memory mapped I/O for embedded systems. This is used when an object must exist at a specific memory address in order for its members to be mapped to external sensors or control pins. Another use case is for memory pools, when the application is responsible for managing its memory instead of relying on the heap.

The syntax for placement new is:

```
unsigned char memory[sizeof(Foo)];
Foo* foo_ptr = new (memory) Foo();
```

In this example, `Foo` was constructed into the `memory` on the stack, instead of on the heap. It is important to understand that using placement new requires the developer to do its own memory management. `memory` must be large enough to contain a `Foo`, and you should never call `delete` on the pointer returned by placement new.

> **Delete**
> Remember that `delete` calls the destructor, and has the heap give memory back to the operating system. However, the `memory` here isn't on the heap! Calling `delete` on `foo_ptr` will cause a crash.

If you think of placement new as simply a constructor call, then it follows that a "placement delete" would simply be a destructor call. And an explicit destructor call is exactly how to clean up after a placement new:

```
foo_ptr->~Foo();
```

You must not forget this step; it isn't a memory leak, but it would be a resource leak.

Here is a more detailed (runnable) example:

```
1 #include <new>
2 #include <iostream>
3 #include <cstdio>
4
5 struct Foo
6 {
7    Foo(void) : value(0)
8    {
9       std::cout << "Constructor" << std::endl;
```

```
10    }
11
12    ~Foo(void)
13    {
14       std::cout << "Destructor" << std::endl;
15    }
16
17    int value;
18 };
19
20 int main(int argc, char* argv[])
21 {
22    // Increase scope
23    {
24       // Allocate memory on the stack
25       unsigned char memory[sizeof(Foo)];
26
27       // Construct a Foo inside that memory
28       Foo* foo_ptr = new (memory) Foo();
29
30       // Show that the memory addresses are the same
31       printf("0x%016lX\n", memory);
32       printf("0x%016lX\n", foo_ptr);
33
34       // Call the destructor explicitly
35       foo_ptr->~Foo();
36    }
37
38    // Memory has been deallocated from the stack
39 }
```

This outputs:

```
Constructor
0x00007FFFFFFFE280
0x00007FFFFFFFE280
Destructor
```

Note that the memory isn't required to be allocated on the stack; you can just as easily allocate the memory using malloc, and later deallocate the memory with free. In fact, you can think of the regular `new` call as performing the following actions:

```
1  #include <new>
2  #include <iostream>
3  #include <cstdio>
4
5  struct Foo
6  {
7     Foo(void) : value(0)
8     {
9        std::cout << "Constructor" << std::endl;
10    }
11
12    ~Foo(void)
13    {
14       std::cout << "Destructor" << std::endl;
15    }
16
17    int value;
18 };
19
20 int main(int argc, char* argv[])
21 {
22
23    // Allocate memory on the heap
24    unsigned char* memory = static_cast<unsigned char*>(malloc(sizeof(Foo)));
25
26    // Construct a Foo inside that memory
27    Foo* foo_ptr = new (memory) Foo();
28
29    // Show that the memory addresses are the same
30    printf("0x%016lX\n", memory);
31    printf("0x%016lX\n", foo_ptr);
32
33    // Call the destructor explicitly
```

```
34    foo_ptr->~Foo();
35
36    // Deallocate the memory from the heap
37    free memory;
38 }
```

This outputs:

```
Constructor
0x0000000000664370
0x0000000000664370
Destructor
```

Dumping Memory

As I said earlier, placement new requires the developer to manage their own memory even more than usual. When working with memory on a low level like this, it becomes very useful to see a hex dump similar to the one created by GDB. With some inspiration[1], I created a function that would pretty-print memory to an output stream:

```
1  void dump_memory(void* ptr,
2                   std::size_t size,
3                   std::ostream& os = std::cout)
4  {
5     typedef unsigned char byte;
6     typedef unsigned long uint;
7
8     // Allow direct arithmetic on the pointer
9     uint iptr = reinterpret_cast<uint>(ptr);
10
11    os << "--------------------------------------------------------------------------\n";
12    os << boost::format("%d bytes") % size;
13
14    // Get number of digits
15    uint indent = std::log10(size) + 1;
16
17    // Write the address offsets along the top row
18    // Account for the indent of "X bytes"
19    os << std::string(13 - indent, ' ');
20    for(std::size_t i = 0; i < 16; ++i)
21    {
22       if(i %  4 == 0){os << " ";}        // Spaces between every 4 bytes
23       os << boost::format(" %2hhX") % i; // Write the address offset
24    }
25
26    // If the object is not aligned
27    if(iptr % 16 != 0)
28    {
29       // Print the first address
30       os << boost::format("\n0x%016lX:") % (iptr & ~15);
31
32       // Indent to the offset
33       for(std::size_t i = 0; i < iptr % 16; ++i)
34       {
35          os << "   ";
36          if(i % 4 == 0){os << " ";}
37       }
38    }
39
40    // Dump the memory
41    for(std::size_t i = 0; i < size; ++i, ++iptr)
42    {
43       // New line and address every 16 bytes, spaces every 4 bytes
44       if(iptr % 16 == 0){os << boost::format("\n0x%016lX:") % iptr;}
45       if(iptr %  4 == 0){os << " ";}
46
47       // Write the address contents
48       os << boost::format(" %02hhX")
49             % static_cast<uint>(*reinterpret_cast<byte*>(iptr));
50    }
51
52    os << "\n--------------------------------------------------------------------------"
```

```
53       << std::endl;
54 }
```

Here is an example of how to use it:

```
1  struct Test
2  {
3     char  a; // 1 byte
4     int   b; // 4 bytes
5     short c; // 2 bytes
6     long  d; // 8 bytes
7
8     Test() :
9        a(0x11),
10       b(0x22222222),
11       c(0x3333),
12       d(0x4444444444444444)
13    {
14
15    }
16 };
17
18 int main(int argc, char* argv[])
19 {
20    unsigned char memory[sizeof(Test)];
21    Test* ptr = new (memory) Test();
22
23    dump_memory(ptr, sizeof(Test));
24
25    ptr->~Test();
26 }
```

And here is the output:

```
------------------------------------------------------------------------
24 bytes              0  1  2  3   4  5  6  7   8  9  A  B   C  D  E  F
0x00007FFFFFFFE270:  11 FF FF FF  22 22 22 22  33 33 00 00  00 00 00 00
0x00007FFFFFFFE280:  44 44 44 44  44 44 44 44
------------------------------------------------------------------------
```

Padding
If you are surprised by the output above, you may not be aware of padding. The compiler will add padding bytes between members in a structure to ensure that each member starts on a proper byte boundary. This means structures may take up more space than if they were packed tightly.

Primitive types in a structure will be padded such that they are aligned on byte boundaries that match their size[2]:

- A char (one byte) will be 1-byte aligned.
- A short (two bytes) will be 2-byte aligned.
- An int (four bytes) will be 4-byte aligned.
- A long (eight bytes) will be 8-byte aligned.
- A float (four bytes) will be 4-byte aligned.
- A double (eight bytes) will be 8-byte aligned.

When working with memory pools, the allocated memory will typically be larger than the object itself. It becomes useful to “mark” the memory with special byte patterns for ease of debugging. I like to use:

- 0xCC for “Clear”
- 0xAA for “Allocated”
- 0xDD for “Deallocated”

For example, lets allocate more memory than we need, and construct the object in the middle of the

array. The markers will indicate what is happening:

```
1  int main(int argc, char* argv[])
2  {
3     // Get the size of our structure
4     unsigned int size = sizeof(Test);
5
6     // Reserve more memory than we need
7     unsigned char memory[size * 2];
8
9     // Fill that memory up with "C" for "Cleared"
10    memset(memory, 0xCC, sizeof(memory));
11
12    // Determine where to construct the object
13    unsigned int offset = size / 2;
14
15    // Mark that area as "Allocated"
16    memset(memory + offset, 0xAA, size);
17
18    // Construct an object offset into memory
19    Test* ptr = new (memory + offset) Test();
20
21    // Lets see what it looks like
22    dump_memory(ptr, sizeof(Test));
23
24    // Destroy the object
25    ptr->~Test();
26 }
```

Outputs:

```
------------------------------------------------------------------------
24 bytes              0  1  2  3   4  5  6  7   8  9  A  B   C  D  E  F
0x00007FFFFFFFE220:                                          11 AA AA AA
0x00007FFFFFFFE230:  22 22 22 22  33 33 AA AA  AA AA AA AA  44 44 44 44
0x00007FFFFFFFE240:  44 44 44 44
------------------------------------------------------------------------
```

A couple things to notice here:

- Only the memory for the object is printed, that is why we don't see any `0xCC`
- The object was offset 12 bytes into the memory
- The padding bytes are filled with `0xAA` as expected

It would be kind of nice to see the memory around the object, to give us some context. One solution is to dump the `memory` variable instead of the `ptr` variable, but with large memory pools this can be too much to look at. Instead, lets just print some additional local context:

```
1  void dump_memory_with_context(void* ptr,
2                                std::size_t size,
3                                std::ostream& os = std::cout)
4  {
5     // Allow direct arithmetic on the pointer
6     unsigned long sptr = reinterpret_cast<unsigned long>(ptr); // Start pointer
7     unsigned long eptr = sptr + size;                          // End pointer
8
9     sptr &= ~15; // Round down to the last multiple of 16
10    sptr -=  16; // Step back one line for context
11    eptr &= ~15; // Round down to the last multiple of 16
12    eptr +=  32; // Step forward one line for context
13
14    // Dump memory
15    dump_memory(reinterpret_cast<void*>(sptr), eptr - sptr, os);
16 }
17
18 int main(int argc, char* argv[])
19 {
20    // Get the size of our structure
21    unsigned int size = sizeof(Test);
22
23    // Reserve more memory than we need
```

```
24    unsigned char memory[size * 2];
25
26    // Fill that memory up with "C" for "Cleared"
27    memset(memory, 0xCC, sizeof(memory));
28
29    // Determine where to construct the object
30    unsigned int offset = size / 2;
31
32    // Mark that area as "Allocated"
33    memset(memory + offset, 0xAA, size);
34
35    // Construct an object offset into memory
36    Test* ptr = new (memory + offset) Test();
37
38    // Lets see what it looks like
39    dump_memory_with_context(ptr, sizeof(Test));
40
41    // Destroy the object
42    ptr->~Test();
43 }
```

Now we get:

```
------------------------------------------------------------------------
80 bytes              0  1  2  3   4  5  6  7   8  9  A  B   C  D  E  F
0x00007FFFFFFFE210:  A0 E2 FF FF  FF 7F 00 00  6C 6B 40 00  00 00 00 00
0x00007FFFFFFFE220:  CC CC CC CC  CC CC CC CC  CC CC CC CC  11 AA AA AA
0x00007FFFFFFFE230:  22 22 22 22  33 33 AA AA  AA AA AA AA  44 44 44 44
0x00007FFFFFFFE240:  44 44 44 44  CC CC CC CC  CC CC CC CC  CC CC CC CC
0x00007FFFFFFFE250:  88 E3 FF FF  FF 7F 00 00  75 94 42 00  01 00 00 00
------------------------------------------------------------------------
```

This shows a local context memory dump that includes our object and the memory around it. Our markers can be clearly seen, as well as the object itself. In addition, we see some other garbage from the stack.

Alignment

In the last section, we constructed an object offset into our memory area, and the result was misaligned in the stack. This situation is not ideal.

Memory alignment is important because while programmers think in terms of bytes, CPUs think in terms of words. On a 64-bit system, this means the CPU will load 8 bytes at a time from memory. If you have an unaligned member that straddles two words, the CPU needs to make twice as many memory accesses as it would have if the memory was properly aligned.[3]

GCC will typically handle memory alignment for you when allocating from the heap, but in the case of a memory pool it is up to the developer to handle it properly. There are two rules to follow when aligning memory addresses:

- The alignment boundary must be greater or equal to the size of a pointer
- The alignment boundary must be a power of 2

One fast and generic method for aligning memory is to allocate a little bit more than needed in order to get to the next boundary, then store the original pointer in the space that was skipped over. It is important to keep the original pointer around because it will be needed when it is time to free that memory.

Below is a modification of an alignment algorithm found in the Eigen[4] library. The following enhancements were made:

- Allow for user specified alignment
- Add compile time checks for alignment rules

- Provide a method for accessing the unaligned pointer

```
#include <boost/mpl/assert.hpp>
#include <boost/mpl/int.hpp>
#include <boost/mpl/comparison.hpp>
#include <boost/mpl/bitwise.hpp>
#include <boost/mpl/arithmetic.hpp>

template<std::size_t bytes = 0>
struct boundary
{
   // If an alignment is specified, it must be greater than or equal to
   // the size of a pointer.
   BOOST_MPL_ASSERT((
      boost::mpl::greater_equal<
         boost::mpl::int_<bytes>,
         boost::mpl::int_<sizeof(void*)>
      >
   ));

   // The alignment bytes must be a power of two
   // (n & (n-1)) == 0
   BOOST_MPL_ASSERT((
      boost::mpl::equal_to<
         boost::mpl::bitand_<
            boost::mpl::int_<bytes>,
            boost::mpl::minus<
               boost::mpl::int_<bytes>,
               boost::mpl::int_<1>
            >
         >,
         boost::mpl::int_<0>
      >
   ));

   // In order for this to work, must allocate additional bytes
   enum{value = bytes};

   // Get the next aligned pointer
   static void* next(void* ptr)
   {
      // Round down to the previous multiple of X,
      // then move to the next multiple of X.
      return
          reinterpret_cast<void*>(
              (reinterpret_cast<std::size_t>(ptr)
                  & ~(std::size_t(value - 1)))
              + value);
   }

   // Return an aligned pointer
   static void* align(void* ptr)
   {
      // Get the next aligned pointer
      void* aligned_ptr = next(ptr);

      // Save the original pointer in the space we skipped over
      *(reinterpret_cast<void**>(aligned_ptr) - 1) = ptr;

      // Return the aligned pointer
      return aligned_ptr;
   }

   // Retrieve the original pointer
   static void* unalign(void* ptr)
   {
      return *(reinterpret_cast<void**>(ptr) - 1);
   }
};

// Specialize to not attempt alignment
template<>
struct boundary<0>
{
   enum{value = 0};
   static void* next   (void* ptr){return ptr;}
   static void* align  (void* ptr){return ptr;}
   static void* unalign(void* ptr){return ptr;}
```

```
77 };
```

Using it is simple: Any pointer can be aligned on a boundary, and any aligned pointer can be unaligned. However, it is important to remember that extra space must be allocated before attempting to align the pointer. The following example will show how to align a structure on a 16 byte boundary:

```
1  int main(int argc, char* argv[])
2  {
3     // Get the size of our structure
4     unsigned int size = sizeof(Test);
5
6     // Purposely offset the object by 8 bytes
7     unsigned int offset = 8;
8
9     // Reserve size for our structure
10    // plus alignment overhead
11    // plus our test offset
12    unsigned char* memory = static_cast<unsigned char*>(malloc(size + boundary<16>::value + offset));
13
14    // Print out the address of the malloc'd memory pointer
15    printf("Memory:          0x%016lX\n", memory);
16
17    // Fill that memory up with "C" for "Cleared"
18    memset(memory, 0xCC, sizeof(memory));
19
20    // Mark the object area as "Allocated"
21    memset(memory + offset, 0xAA, size + boundary<16>::value);
22
23    // Print out the address of where we will construct the object
24    printf("Memory + Offset: 0x%016lX\n", memory + offset);
25
26    // Construct an object offset into memory,
27    // but use pointer alignment to realign it
28    Test* ptr = new (boundary<16>::align(memory + offset)) Test();
29
30    // Print out the address of the aligned pointer
31    printf("Ptr:             0x%016lX\n", static_cast<void*>(ptr));
32
33    // Print out the address of the unaligned pointer
34    printf("Unaligned Ptr:   0x%016lX\n", boundary<16>::unalign(ptr));
35
36    // Lets see what it looks like
37    dump_memory_with_context(ptr, sizeof(Test));
38
39    // Destroy the object
40    ptr->~Test();
41
42    // Unalign the aligned pointer to get the original pointer back
43    // then remove the offset to properly free
44    free(boundary<16>::unalign(ptr) - offset);
45 }
```

This outputs:

```
Memory:          0x00000000006648F0
Memory + Offset: 0x00000000006648F8
Ptr:             0x0000000000664900
Unaligned Ptr:   0x00000000006648F8
------------------------------------------------------------------------
64 bytes              0  1  2  3   4  5  6  7   8  9  A  B   C  D  E  F
0x00000000006648F0:  CC CC CC CC  CC CC CC CC  F8 48 66 00  00 00 00 00
0x0000000000664900:  11 AA AA AA  22 22 22 22  33 33 AA AA  AA AA AA AA
0x0000000000664910:  44 44 44 44  44 44 44 44  AA AA AA AA  AA AA AA AA
0x0000000000664920:  4B 00 00 00  00 00 00 00  31 00 00 00  00 00 00 00
------------------------------------------------------------------------
```

So what happened here? Here is the output color-coded:

```
Memory:           0x00000000006648F0
Memory + Offset:  0x00000000006648F8
Ptr:              0x0000000000664900
Unaligned Ptr:    0x00000000006648F8
-------------------------------------------------------------------------
64 bytes                0  1  2  3   4  5  6  7   8  9  A  B   C  D  E  F
0x00000000006648F0:   CC CC CC CC  CC CC CC CC  F8 48 66 00  00 00 00 00
0x0000000000664900:   11 AA AA AA  22 22 22 22  33 33 AA AA  AA AA AA AA
0x0000000000664910:   44 44 44 44  44 44 44 44  AA AA AA AA  AA AA AA AA
0x0000000000664920:   4B 00 00 00  00 00 00 00  31 00 00 00  00 00 00 00
-------------------------------------------------------------------------
```

Memory Dump

- Red: The debug markers
- Orange: The 16-byte aligned pointer address
- Yellow: The test object
- Green: The original unaligned / offset pointer address
- Blue: The beginning of the memory

Now, remember that we offset the object 8 bytes into the beginning of the memory, which would be `0x006648F8`. The object was then aligned to the next 16-byte boundary, starting at `0x00664900` (16-byte aligned addresses always end in 0). In order to do this, an extra 16 bytes needed to be allocated, which can be seen with the red `0xAA` markers. Finally, the original pointer of `0x006648F8` was saved immediately before the aligned pointer in little endian (which in this case happens to have the same address as itself).

> **Endianness**
> Endianness is the ordering of bytes in a word (On a 64-bit machine, there are 8 bytes in a word)[5]. The bytes can be ordered in one of two directions:
>
> - Big Endian: The most significant byte is stored at the smallest memory address (the "Big End" first)
> - Little Endian: The least significant byte is stored at the smallest memory address (the "Little End" first)
>
> Big Endian is the more intuitive method, as it matches reading left to right. It is chosen as the standard for transmitting words over a network, and is also known as "Network Byte Order".
>
> Little Endian is more difficult for a human to read, but it has performance advantages and desirable properties. For example, a 32-bit memory location with content `4A 00 00 00` can be read at the same address as either 8-bit (value = `0x4A`), 16-bit (`0x004A`), or 32-bit (`0x0000004A`), all of which retain the same numeric value. Intel's x86/x64 architecture is little endian.
>
> It is important to be aware that endianness doesn't affect byte arrays or strings.

Here is what the memory would look like if we did the alignment without the forced 8 byte offset:

```
Memory:           0x0000000000664370
Memory + Offset:  0x0000000000664370
Ptr:              0x0000000000664380
Unaligned Ptr:    0x0000000000664370
-----------------------------------------------------------------------
64 bytes                0  1  2  3   4  5  6  7   8  9  A  B   C  D  E  F
0x0000000000664370:    AA AA AA AA  AA AA AA AA  70 43 66 00  00 00 00 00
0x0000000000664380:    11 AA AA AA  22 22 22 22  33 33 AA AA  AA AA AA AA
0x0000000000664390:    44 44 44 44  44 44 44 44  31 00 00 00  00 00 00 00
0x00000000006643A0:    00 43 00 00  00 00 00 00  00 00 00 00  00 00 00 00
-----------------------------------------------------------------------
```

Wasted Space

As you can see, even though the memory was already aligned, 16 bytes were wasted in order to get to the next aligned address. You can also see the original pointer address stored immediately before the aligned address. If we were doing 8 byte alignment, the extra 8 bytes that are wasted to align on the next boundary would contain the original address. Attempting to perform an alignment less than 8 means that there wouldn't be enough space for the original pointer; however, one of the rules of alignment states that the alignment must be greater than or equal to the size of a pointer, so we don't need to worry about that case.

What about larger alignments? Here is what it would look like if we were doing 32-byte alignment with an 8 byte offset:

```
Memory:           0x0000000000664820
Memory + Offset:  0x0000000000664828
Ptr:              0x0000000000664840
Unaligned Ptr:    0x0000000000664828
-----------------------------------------------------------------------
80 bytes                0  1  2  3   4  5  6  7   8  9  A  B   C  D  E  F
0x0000000000664820:    CC CC CC CC  CC CC CC CC  AA AA AA AA  AA AA AA AA
0x0000000000664830:    AA AA AA AA  AA AA AA AA  28 48 66 00  00 00 00 00
0x0000000000664840:    11 AA AA AA  22 22 22 22  33 33 AA AA  AA AA AA AA
0x0000000000664850:    44 44 44 44  44 44 44 44  AA AA AA AA  AA AA AA AA
0x0000000000664860:    00 00 00 00  00 00 00 00  51 00 00 00  00 00 00 00
-----------------------------------------------------------------------
```

Large Alignment

Generally, 32-byte alignment is unnecessary. There are cases where 16-byte alignment is needed though, such as when dealing with the SSE instructions on the CPU (for example, matrix multiplication with the Eigen library).

---

1. Memory Pool Tutorial – Marco Alamia↩
2. Data Structure Alignment↩
3. Purpose of Memory Alignment↩
4. Eigen Handmade Aligned Malloc – Benoit Jacob↩
5. Endianness↩

Posted by Joe Ruether Aug 23rd, 2015 C++, Memory

About The Author

Joe Ruether is the lead software engineer for the Multisensor Aircraft Tracking system at SRC Inc. He is an expert in C++ template metaprogramming and is interested in cryptography and open source software.

Follow @jrruethe

« Yaml De/Serialization with Boost Fusion Setting up Qubes »

Comments

Disqus seems to be taking longer than usual. Reload?

Contact

jrruethe@gmail.com
keybase.io/jrruethe

Public Key:
4F40 99F8 276B DBA5 475A
8446 4630 BEDC 40B9 35FE

Follow @jrruethe

Link to this page

Related Posts

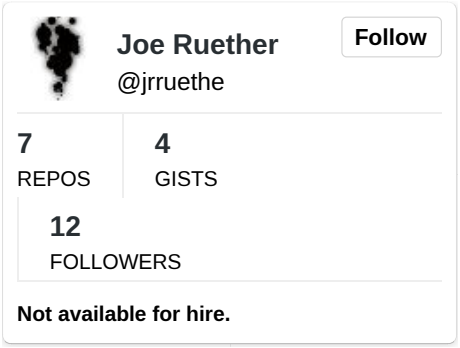
- Singletons

- Boost Fusion Json Serializer
- Object Counter
- Yaml De/Serialization with Boost Fusion
- Object Pool

Recent Posts

- Cryptography Using OpenSSL
- Running Docker in Qubes
- Bitcoin Donation Proofs
- Bitcoin Donations
- Object Pool

GitHub Repos



- juice

  Juice Oracle

- wordlist_filter
- myretail
- basex

  Convert binary to various text representations

- json_encoder

  Encode JSON into a QR code

Bitcoin Donations

Bitcoin
187gRAhdyD2KaAhMN
D92RQMqQQMQtW678m
Proof

$41597.25

High: $41709.63 Low: $41498.9

Powered by CoinDesk