Payload Type

The following notes illustrate debugging a linked list implementation with gdb.

The example makes use of the following payload type:

```c
struct _WordRecord {
    char*    Word;    // zero-terminated C-string
    uint32_t Freq;    // number of occurrences in
};
typedef struct _WordRecord WordRecord;

void     WordRecord_Init(WordRecord* const pWR,
                          const char* const pWord, uint32_t Frequency);
char*    WordRecord_Word(const WordRecord* const pWR);
uint32_t WordRecord_Frequency(const WordRecord* const pWR);
bool     WordRecord_Increment(WordRecord* const pWR);
bool     WordRecord_Decrement(WordRecord* const pWR);
bool     WordRecord_equals(const WordRecord* const pLeft,
                          const WordRecord* const pRight);
void     WordRecord_Clear(WordRecord* const pWR);
void     WordRecord_Print(const WordRecord* const pWR,
                          FILE *fp);
```

The source code for the payload type is probably available on the course website.
"Duct-tape" Wrapper

The example uses the following "duct-tape" wrapper type to attach payload values to list nodes:

```c
struct _WordRecordDT {
    WordRecord WR;
    DNode node;
};

typedef struct _WordRecordDT WordRecordDT;

void WordRecordDT_Init(WordRecordDT* const pLE, const WordRecord* const pWR);
```
List Interface

The list is fully generic and uses the following interface (which you've seen before):

```c
struct _DNode {
    struct _DNode *prev;    // pointer to previous list element
    struct _DNode *next;    // pointer to next list element
};

struct _DList {
    struct _DNode head;    // List head sentinel
    struct _DNode tail;    // List tail sentinel
};

typedef struct _DNode DNode;
typedef struct _DList DList;

#define DList_Entry(LIST_ELEM, STRUCT, MEMBER)         
    ((STRUCT *) ((uint8_t *) (LIST_ELEM) – 
        offsetof (STRUCT, MEMBER)))
```

...
List Interface

The list is fully generic and uses the following interface:

```c
... void DList_Init(DList* pL);
void DList_PushFront(DList* pL, DNode* node);
void DList_PushBack(DList* list, DNode* elem);
void DList_Insert(DNode* before, DNode* elem);

DNode* DList_Remove(DNode* elem);
DNode* DList_PopFront(DList* list);
DNode* DList_PopBack(DList* list);

bool DList_Empty(DList* list);
... 
```
List Interface

The list is fully generic and uses the following interface:

```c
DNode* DList_Begin(DList* pL);
DNode* DList_Next(DNode* elem);
DNode* DList_End(DList* list);
DNode* DList_RBegin(DList* list);
DNode* DList_Prev(DNode* elem);
DNode* DList_REnd(DList* list);
DNode* DList_Head(DList* list);
DNode* DList_Tail(DList* list);
DNode* DList_Front(DList* list);
DNode* DList_Back(DList* list);
```
Here is one version of the driver code:

```c
.. .
int main() {

    char* Words[NUMWORDS];
    Words[0] = "zero";
    Words[1] = "one";
    Words[2] = "two";
    Words[3] = "three";
    Words[4] = "four";

    DList myList;
    DList_Init(&myList);

    .. .
```
Here is one version of the driver code:

```c
... 
WordRecord WR;

for (int i = 0; i < NUMWORDS; i++) {
    WordRecordDT *pWRDT = malloc(sizeof(WordRecordDT));
    assert( pWRDT != NULL );

    WordRecord_Init(&WR, Words[i], i);
    WordRecordDT_Init(pWRDT, &WR);

    DList_PushBack(&myList, &pWRDT->node);
}

printList(&myList, stdout);
fprintf(stdout, "\n");

return 0;
}
```
We'll compile the code for examination under gdb:

```
gcc -o driver -std=c99 -Wall -O0 -static -m32 –ggdb3 driver.c WordRecord.c WordRecordDT.c DList.o
```

Note: I'm posting an object file for the implementation of DList.

Start a `gdb` session; some uninformative output has been clipped and some formatting has been altered for clarity.

```
2001: session1 > gdb driver
.
(gdb) break driver.c:22
Breakpoint 1 at 0x80482f1: file driver.c, line 22.
(gdb) run
Starting program:
/home/williammcquain/2505/gdb/DList/session1/driver

Breakpoint 1, main () at driver.c:23
23       DList_Init(&myList);
.
```
... Breakpoint 1, main () at driver.c:23
23 DList_Init(&myList);
(gdb) print Words

$1 = {0x80af348 "zero",
     0x80af34d "one",
     0x80af351 "two",
     0x80af355 "three",
     0x80af35b "four"}

OK, we've reached the breakpoint; let's examine the array Words[]

How would you explain the data shown here?
$1 = \{0x80af348 \ "zero",
0x80af34d \ "one",
0x80af351 \ "two",
0x80af355 \ "three",
0x80af35b \ "four}\}

Do those pointer values make sense?
Remember how strings are stored in C.

These are the values stored in the array Words[].
So, these are pointers to the strings.
Examining Execution

... 
Breakpoint 1, main () at driver.c:23
23           DList_Init(&myList);
...
(gdb) n

OK, let's execute the call to DList_Init()

27        for (int i = 0; i < NUMWORDS; i++) {

(gdb) print myList
$2 = {head = {prev = 0xffffffffd800, next = 0xffffffffd808},
     tail = {prev = 0xffffffffd800, next = 0xffffffffd808}}
...

And, examine the DList variable

How can we determine whether these values make sense?
Examining Execution

```
(gdb) print myList
$2 = {head = {prev = 0xfffffd800, next = 0xfffffd808},
     tail = {prev = 0xfffffd800, next = 0xfffffd808}}

We can display the addresses of myList.head and myList.tail

(gdb) print &myList.head
$3 = (struct _DNode *) 0xfffffd800

(gdb) print &myList.tail
$4 = (struct _DNode *) 0xfffffd808

Now... do the contents of myList appear to be correct?
```
(gdb) print myList
$2 = {head = {prev = 0xfffffd800, next = 0xfffffd808},
  tail = {prev = 0xfffffd800, next = 0xfffffd808}}

(gdb) print &myList.head
$3 = (struct _DNode *) 0xfffffd800

(gdb) print &myList.tail
$4 = (struct _DNode *) 0xfffffd808

Yep. Compare the addresses of the two node objects to the values of their prev and next pointers.
Let's create our first WordRecord and WordRecordDT objects:

```c
for (int i = 0; i < NUMWORDS; i++) {
...
(gdb) n
28     WordRecordDT *pWRDT = malloc( sizeof(WordRecordDT) );
(gdb) n
29     assert( pWRDT != NULL );
(gdb) n
31     WordRecord_Init(&WR, Words[i], i);
(gdb) n
...
(gdb) print WR
$5 = {Word = 0x80d42b0 "zero", Freq = 0}
...
```

Is the WordRecord object as expected?
Examining Execution

. . .
32         WordRecordDT_Init(pWRDT, &WR);
(gdb) n
. . .

(gdb) print *pWRDT
$6 = {WR = {Word = 0x80d42c0 "zero", Freq = 0},
      node = {prev = 0x0, next = 0x0}}
. . .

(gdb) print &(pWRDT->WR)
$9 = (WordRecord *) 0x80d42d0

(gdb) print &(pWRDT->node)
$10 = (DNode *) 0x80d42d8
. . .

OK, we've initialized a wrapper object. Let's examine its structure:

Let's see where the parts are stored in memory:

What does that tell us?
(gdb) print & (pWRDT->WR)
$9 = (WordRecord *) 0x80d42d0

(gdb) print & (pWRDT->node)
$10 = (DNode *) 0x80d42d8

This tells us the memory layout of this WordRecordDT object:

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80D42D0</td>
<td>WordRecord member</td>
</tr>
<tr>
<td>0x80D42D8</td>
<td>DNode member</td>
</tr>
</tbody>
</table>

And, therefore, the layout of every WordRecordDT object:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>WordRecord member</td>
</tr>
<tr>
<td>8</td>
<td>DNode member</td>
</tr>
</tbody>
</table>
Let's insert the first object into the list:

And examine myList now:

And see what head.next points to:

Does that look OK?
Now we'll get sneaky. Here's a nice use of pointer arithmetic and typecasting to access the payload relative to the node object within the wrapper:

(gdb) print *(WordRecord*)((uint8_t*)myList.head.next-8)
$16 = {Word = 0x80d42c0 "zero", Freq = 0}

Does that look OK?
Fine Points

(gdb) print *(WordRecord*)((uint8_t*)myList.head.next - 8)

$16 = \{\text{Word} = 0x80d42c0 \ "zero", \text{Freq} = 0\}

The pointer manipulation logic is:

myList.head.next points to the DNode object within a wrapper object
But… that pointer is of type DNode*

(uint8_t*) myList.head.next also points to that DNode object
But… this (nameless) pointer has a 1-byte target

(uint8_t*) myList.head.next – 8 points to the beginning of the wrapper
And hence to the WordRecord object within the wrapper
But… that pointer has a 1-byte target

(WordRecord*) ((uint8_t*) myList.head.next – 8) has a WordRecord target

*(WordRecord*) ((uint8_t*) myList.head.next – 8) is that target!
Summary

OK, now you know how to do the following things:

- display and interpret the contents of struct variables
- display and interpret pointer values
- display and interpret targets of pointers
- use C syntax in specifying pointer expressions
- apply pointer arithmetic
Now we will add the following loop to the original driver:

```c
... 

while ( !DList_Empty(&myList) ) {
    DNode* current = DList_PopBack(&myList);
    WordRecordDT *pWRDT = DList_Entry(current, WordRecordDT, node);
    WordRecord_Print(&pWRDT->WR, stdout);
    fprintf(stdout, "\n");
}

return 0;
```
Error

When we run the driver, it should create and display the contents of a list with 5 data nodes. Which it does.

Then, it should delete the last data node and display the corresponding payload until the list is empty.

What it actually does is fall into an infinite loop, repeatedly printing the same value.

We'll examine this behavior in gdb…
Examining Execution

We'll set a breakpoint at the new while loop and run to that line:

```c
2031: session2 > gdb driver
... (gdb) break driver.c:40
Breakpoint 1 at 0x80483ca: file driver.c, line 40.
(gdb) run
Starting program:
/home/williammcquain/2505/gdb/DList/session2/driver
  0: zero
  1: one
  2: two
  3: three
  4: four

OK, the list that was created by the first loop displayed correctly... that's something at least...

Breakpoint 1, main () at driver.c:40
40 while ( !DList_Empty(&myList) ) {
  ...
```
40 while ( !DList_Empty(&myList) ) {
(gdb) n
42 DNode* current = DList_PopBack(&myList);
(gdb) n
43 WordRecordDT *pWRDT = DList_Entry(current, WordRecordDT, node);
(gdb) n
44 WordRecord_Print(&pWRDT->WR, stdout);
(gdb) n
45 fprintf(stdout, "\n");

(gdb) n
4: four
40 while ( !DList_Empty(&myList) ) {
.
.
OK, the correct value was displayed after popping the original final record from the list.
We'll set a breakpoint at the new while loop and run to that line:

```
40     while ( !DList_Empty(&myList) ) {

Look at the contents of myList; doesn't tell us much...

(gdb) print myList
$1 = {head = {prev = 0xfffffd7f8, next = 0x80d42a0},
    tail = {prev = 0x80d4380, next = 0xffffd800}}

Look at the preceding node; next is OK...

(gdb) print *myList.tail.prev
$2 = {prev = 0x80d4348, next = 0xffffd800}

(gdb) print *(WordRecord*)((uint8_t*)myList.tail.prev-8)
$3 = {Word = 0x80d43a0 "four", Freq = 4}

...}
```
We'll set a breakpoint at the new while loop and run to that line:

```gdb
(gdb) print *myList.tail.prev
$2 = {prev = 0x80d4348, next = 0xffffffd800}
```

Let's check the current last payload:

```gdb
(gdb) print *(WordRecord*)((uint8_t*)myList.tail.prev-8)
$3 = {Word = 0x80d43a0 "four", Freq = 4}
```

Well, that's not good... that should have been gone.

Looks like PopBack() didn't do its job.
DList_PopBack()

DNode* DList_PopBack(DList* list) {
    ... DNode* back = DList_Back(list);
    DList_Remove (back);
    return back;
}

That looks OK, as far as it goes, but this depends on the implementations of two other DList functions.

DNode* DList_Remove(DNode* elem) {
    ... elem->prev->next = elem->next;
    return elem->next;
}

That looks suspicious. Shouldn't this be resetting two pointers instead of one?

You'd better check the logic.