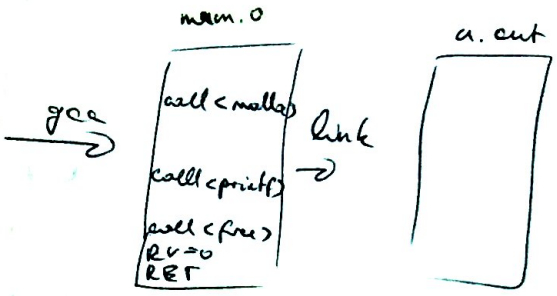
CS 107 lecture 13


```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
int main()
{
    void *mem = malloc(400);
    assert(mem != NULL);
    printf("Yay!\n");
    free(mem);
    return 0;
}
```



if `#include <stdio.h>` commented out

gcc will still compile and infer prototype of printf:

By default when a prototype is inferred, int return [redacted] is inferred, with warnings

↳ since <stdio.h> does not generate any code, the compilation in this case will be the same as if <stdio.h> is included

- gcc allways searches standard libraries for assembly code, even if prototypes are not included.

↑ .o files

all calls will have to have at compile time the same # of [redacted] params

if `#include <stdlib.h>` is commented out

↳ no prototypes for malloc...

1) then infer malloc prototype from malloc(400), taking int and returning int which is assigned to void * pointer

2 warnings ←

2) infer free prototype

↳ 3rd warning

But after compilation, still the same .o file generated, warnings are not stored in .o and linking works fine in these cases

#include
if <assert.h> is commented out

well infer assert in asser(mem != NULL) as
a function taking boolean / int and returning int
but it is a macro, not a function, defined in assert.h

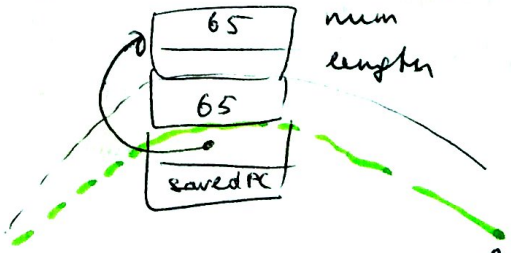
↳ compilation will [redacted] be OK
with CALL <assert>
but linking will fail because there is
no code for assert function.

Prototypes are contracts between caller and callee
for the space above saved PC.

```
int main()
{
    int num = 65;
    int length = strlen((char*) &num, num);
    printf("Length = %d\n", length);
    return 0;
}
```

→ compiled with 1 warning (not prototype of strlen)
.o files have no explicit information
that strlen takes only one parameter
implicit in SP assignments, but not explicit

strlen AR



→ linked OK 0|0|0|65
→ returns 0 on Big endian
and 1 on little endian
65|0|0|0

linked strlen does
not see 65 as parameter
because it expects only 1 parameter

Except one warning, it causes no problems in compilation,
linking and execution but [redacted] unintegrated result

CS 107 Lecture 13


```
int memcmp (void *v1);
```

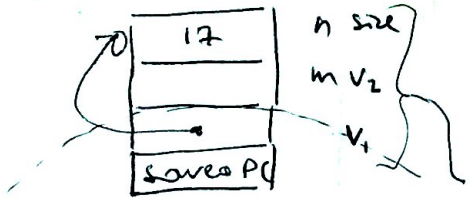
← expects 3 arguments

.o file expects:

```
int memcmp (void *v1,
            void *v2,
            int size)
```

```
{
  int n = 17;
  int m = memcmp (&n);
}
```

AR:



[redacted] after memcmp from .o is linked it accesses 12 bytes as 3 parameters above saved PC

will compile, link, execute, probably crash since m / v2 is uninitialized unless there was a bit pattern that is an address

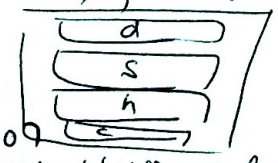
C: CALL <memcmp>

C++: CALL <memcmp_void_p>

attaches automatically → enables function overload

and [redacted] also will lead to linking problems of memcmp example thus preventing the above behaviour

seg fault: * (NULL), dereferencing a bad pointer, but most often due to dereferencing NULL



Null address: 4 bytes at address 0, 1, 2, 3 of memory are not part of any memory segment [redacted]

bus errors: the address is in a segment, but dereferencing it could not be possibly correct

e.g. `void *vp = ...;`
`*(short *)vp = 7;`

[redacted]
all int begin at address multiple of 4
all short begin at even addresses

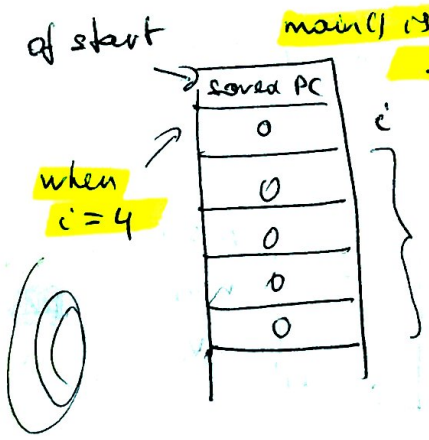
[redacted] char has not restriction

`*(short *)vp = 7;` bus error with some probability (~0.5) ← if vp is odd address

`*(int *)vp = 55;` bus error with some probability of vp is not a multiple of 4

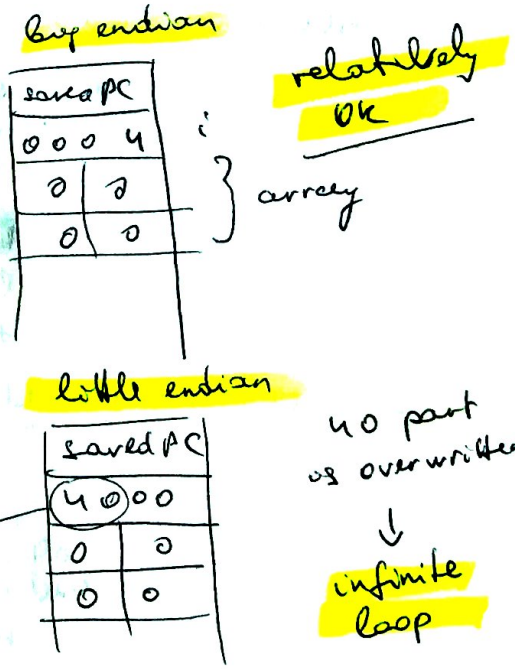
infinite loop

```
int main()
{
  int i;
  int array[4];
  for (i=0; i<=4; i++){
      array[i] = 0;
  }
  return 0;
}
```



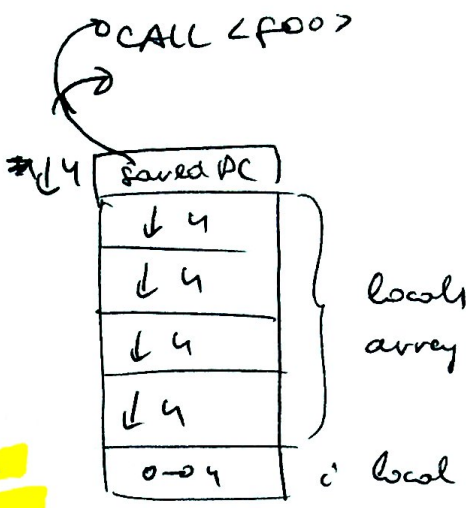

```
int main()
{
  int i;
  short array[4];
  for (i=0; i<=4; i++){
      array[i] = 0;
  }
  return 0;
}
```

my comment
the problem would be different if array was dynamically allocated and then the entire array would not be on stack.



```
void foo()
{
  int array[4];
  int i;
  for (i=0; i<=4; i++) {
      array[i] -= 4;
  }
}
```

infinite calls



all instructions are 4 Byte
saved PC point to instruction
after CALL <foo>