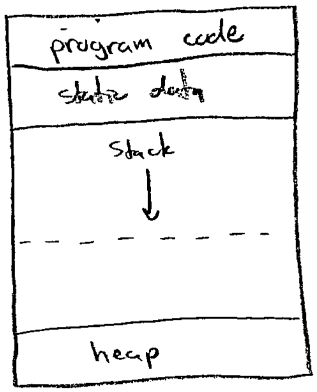
C Program Memory Organization



Program Code: Compiled instructions for executing program

Static Data: Data that persists throughout program execution
(i.e. global variables)

Stack: Temporary memory used for function calls
Note: Data does **NOT** persist after returning from function

Heap: Dynamically allocated memory (e.g. memory allocated by calls to malloc())

* All data (and code) within a C program is stored within a memory at a specific memory location (i.e. **memory address**)

C Function Calls

- The program stack is used to keep track of information during function calls.
- This information is referred to as the activation record. (or stack frame)

Activation Record: include storage for input parameters, return value, temporary storage, saved state information (i.e information used to manage stack and return to location function was called from)

```
#include <stdio.h>

int main() {

   int  val = 10;
   int  mod;


   mod = FuncE(val);
③  printf("%i  %i\n", val, mod);

   return 0;
}
```

```
int FuncE(int input) {
①
   if (input <=10) input += 5;

②  return input-1;
}
```

Stack at ①:

val mod	10 ??
input return value	10 ??
temp. storage return location	③

main's activation record

FuncE's activation record

Stack at ②:

val mod	10 ??
input return value	15 14
temp. storage return location	③

Stack at ③:

val	10
mod	14

Pointers: A C pointer stores the address at which data is located.
↳ e.g. although there are some exceptions, a pointer is simply a memory address

Need methods to: ① operate on pointers themselves
② operate on the data pointed to by the pointer

* what if pointer doesn't point to anything?

```
int a;
int *iptr;
int **iptr_ptr;
```

iptr = &a; ⇒ & returns the address at which the variable a is located, stored in pointer

*iptr = 10; ⇒ * access the item pointed to by the pointer (in this case, 10 is written to the memory location pointed to by the pointer. Thus a is now 10)

iptr_ptr = &iptr; ⇒ pointers are also variables stored in memory, so they also have an address

Array Accesses using Pointers (Pointer Arithmetic):

Pointers are commonly used to access arrays. C supports pointer arithmetic to make doing so easy (but it can be confusing).

- a pointer can be directly assigned to an array. Same as assigning pointer to address of the first element
- incrementing (or decrementing) a pointer will increment (or decrement) the memory location stored to access the next (or previous) location based on the <u>type</u> of data being accessed.

Example:

```
int     test_array[100];
int  sum;
int  *iptr;
   ...
for(iptr=test_array; iptr < &test_array[100]; iptr++) {
      sum +=  *iptr;
}
```

Note: there is an error in the above example:

The following are equivalent:

① iptr = test_array;

② iptr = & test_array[0];

① test_array[0+5] = 42;

② iptr = test_array;
*(iptr+5) = 42;

Dynamic Memory Allocation:

- We often do not know how much memory we will need before executing our program.

- Can dynamically allocate memory as needed using malloc, realloc, calloc, and free

```
void* malloc (size_t size);
void * calloc (size_t size);
void * realloc (void* ptr, size_t size);
void free (void *ptr);
```

- malloc() allocates a block of memory of size bytes in the program heap and returns a pointer to the first memory location if successful. Returns NULL if memory cannot be allocated

  ↳ malloc doesn't know what you are allocating (int, char array, etc.) so it returns a void *. Must be cast to proper pointer type

  ↳ always check the return value.

Typical method of dynamic memory allocation:

```
int *iptr;


iptr = malloc (sizeof (int));
if (iptr == NULL) {
   printf ("could not allocate memory\n");
   exit(-1);
}
```

Pointers to pointers (common in function parameters):

```
int g (int **iptr) {
   *iptr = (int *) malloc (sizeof (int) * 5);
   if ( *iptr == NULL) return -1;
   return 5;
}


int *new_array;
int num-elements;

num-elements = g(&new_array);
if (num-elements >= 0) {
   int i=0;
   int aptr = new_array;
   while (i < num_elements) {
        *aptr = 0;
        aptr ++;
        i++;
   }
}
```