CS 107
Lecture 6: Stack and Heap

Friday, January 29, 2018

Computer Systems
Winter 2018
Stanford University
Computer Science Department

Reading: Reader: Ch 4, C Primer, K&R Ch 1.6, 5.1-5.5

Lecturers: Chris Gregg
Today's Topics

- Logistics
  - Assign2 — Due Monday Jan 29th at 11:59pm for on-time bonus
  - To get line numbers and proper tabbing in vim, run this:
    - `cp /afs/ir/class/cs107/samples/vimrc ~/.vimrc`
- Reading: Reader: *C Primer*
  - Pointers to Arrays (finish from last time)
  - Stack allocation
  - Stack frames
  - Parameter passing
  - Dynamic allocation (malloc/realloc/free).
  - More Pointers to pointers
One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. The lab had a good example: `envp`.

With arrays:
1. Always draw a picture!!!1! Make up the addresses -- the actual numbers aren't particularly important for understanding.
One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. The lab had a good example: `envp`.

With arrays:
1. Always draw a picture!!! Make up the addresses -- the actual numbers aren't particularly important for understanding.
One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. The lab had a good example: `envp`.

With arrays:
1. Always draw a picture!!!1! Make up the addresses -- the actual numbers aren't particularly important for understanding.
One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. The lab had a good example: `envp`.

With arrays:
1. Always draw a picture!!! Make up the addresses -- the actual numbers aren't particularly important for understanding.
One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. The lab had a good example: \texttt{envp}.

With arrays:

1. Always draw a picture!!!1! Make up the addresses -- the actual numbers aren't particularly important for understanding.
One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. The lab had a good example: `envp`.

With arrays:
1. Always draw a picture!!! Make up the addresses -- the actual numbers aren't particularly important for understanding.
2. If you know the type that is held in the array, you can always dereference to get a single pointer to the type. E.g., `envp[0]` is a pointer to the string "XDG_SESSION_ID=3230"

What is the value of `envp[2]` for the diagram?

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x6a70</td>
<td>0x0</td>
<td>SSH_TTY=/dev/pts/29</td>
</tr>
<tr>
<td>0x6a68</td>
<td>0x7d4f</td>
<td>HISTSIZE=5000</td>
</tr>
<tr>
<td>0x6a60</td>
<td>0x7d41</td>
<td>SHELL=/bin/bash</td>
</tr>
<tr>
<td>0x6a58</td>
<td>0x7d31</td>
<td>TERM=xterm-256color</td>
</tr>
<tr>
<td>0x6a50</td>
<td>0x7d1d</td>
<td>XDG_SESSION_ID=3230</td>
</tr>
<tr>
<td>0x6a48</td>
<td>0x7d09</td>
<td></td>
</tr>
</tbody>
</table>
What is the value of `envp[2]` for the diagram?

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x6a70</td>
<td>0x0</td>
<td>SSH_TTY=/dev/pts/29</td>
</tr>
<tr>
<td>0x6a68</td>
<td>0x7d4f</td>
<td>HISTSIZE=5000</td>
</tr>
<tr>
<td>0x6a60</td>
<td>0x7d41</td>
<td>SHELL=/bin/bash</td>
</tr>
<tr>
<td>0x6a58</td>
<td>0x7d31</td>
<td>TERM=xterm-256color</td>
</tr>
<tr>
<td>0x6a50</td>
<td>0x7d1d</td>
<td>XDG_SESSION_ID=3230</td>
</tr>
</tbody>
</table>
```

0x6a50

With arrays:

1. Always draw a picture!!! Make up the addresses -- the actual numbers aren't particularly important for understanding.
2. If you know the type that is held in the array, you can always dereference to get a single pointer to the type. E.g., `envp[0]` is a pointer to the string "XDG_SESSION_ID=3230"
One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. The lab had a good example: envp.

With arrays:
1. Always draw a picture!!!1! Make up the addresses -- the actual numbers aren't particularly important for understanding.
2. If you know the type that is held in the array, you can always dereference to get a single pointer to the type. E.g., envp[0] is a pointer to the string "XDG_SESSION_ID=3230"

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x6a70</td>
<td>0x0</td>
<td>SSH_TTY=/dev/pts/29</td>
</tr>
<tr>
<td>0x6a68</td>
<td>0x7d4f</td>
<td>HISTSIZE=5000</td>
</tr>
<tr>
<td>0x6a60</td>
<td>0x7d41</td>
<td>SHELL=/bin/bash</td>
</tr>
<tr>
<td>0x6a58</td>
<td>0x7d31</td>
<td>TERM=xterm-256color</td>
</tr>
<tr>
<td>0x6a50</td>
<td>0x7d1d</td>
<td>XDG_SESSION_ID=3230</td>
</tr>
</tbody>
</table>

What type is envp[2]?

![Diagram showing memory addresses and values associated with envp array]
One tricky part of CS 107 for many students is getting comfortable with what the memory looks like for pointers to arrays, particularly when the arrays themselves are filled with pointers. The lab had a good example: \texttt{envp}.

With arrays:
1. Always draw a picture!!!1! Make up the addresses -- the actual numbers aren't particularly important for understanding.
2. If you know the type that is held in the array, you can always dereference to get a single pointer to the type. E.g., \texttt{envp[0]} is a pointer to the string "XDG_SESSION_ID=3230"

What type is \texttt{envp[2]}? \texttt{char *}
Note: `envp` is a weird array in that it is null-terminated! Very, very few arrays have this property in C.

Most arrays are passed with another variable that gives their length. For example, we have `argv` and `argc`.
Let's take a look at two more examples (we looked at the first example on Monday)

For the first example, let's look at the array defined as follows:

```c
int values[] = {8,2,7,14,-5,42};
```

What if we wanted to write a swap function for the array (to swap two elements)? We can do it with regular `int` variables:

```c
void swapA(int *arr, int index_x, int index_y) {
    int tmp = *(arr + index_x);
    *(arr + index_x) = *(arr + index_y);
    *(arr + index_y) = tmp;
}
```
For the first example, let's look at the array defined as follows:

```c
int values[] = {8, 2, 7, 14, -5, 42};
```

What if we wanted to write a swap function for the array (to swap two elements)? **Can we do it with memmove?**
For the first example, let's look at the array defined as follows:

```c
int values[] = {8,2,7,14,-5,42};
```

What if we wanted to write a swap function for the array (to swap two elements)? **Can we do it with memmove?** Sure:

```c
void swapB(int *arr, int index_x, int index_y)
{
    int tmp;
    memmove(&tmp, arr + index_x, sizeof(int));
    memmove(arr + index_x, arr + index_y, sizeof(int));
    memmove(arr + index_y, &tmp, sizeof(int));
}
```
What if we wanted to write a swap function for the array (to swap two elements)? Can we do it with `memmove`? Sure:

```c
void swapB(int *arr, int index_x, int index_y)
{
    int tmp;
    memmove(&tmp, arr + index_x, sizeof(int));
    memmove(arr + index_x, arr + index_y, sizeof(int));
    memmove(arr + index_y, &tmp, sizeof(int));
}
```

This works because we know the size of the elements in the array (they are `ints`)

As long as we know the size of the elements, we can always swap (or compare, or whatever) two elements in an array!
What if we wanted to write a swap function for the array (to swap two elements)? **Do we even need the int tmp? In other words, tmp just holds bytes, so...**

```c
void swapB(int *arr, int index_x, int index_y)
{
    int tmp;
    memmove(&tmp, arr + index_x, sizeof(int));
    memmove(arr + index_x, arr + index_y, sizeof(int));
    memmove(arr + index_y, &tmp, sizeof(int));
}
```
What if we wanted to write a swap function for the array (to swap two elements)? **Do we even need the int tmp? In other words, tmp just holds bytes, so...**

```c
void swapC(int *arr, int index_x, int index_y)
{
    char tmp[sizeof(int)];
    memmove(tmp, arr + index_x, sizeof(int));
    memmove(arr + index_x, arr + index_y, sizeof(int));
    memmove(arr + index_y, tmp, sizeof(int));
}
```

If we wanted to, we could just declare an array that we size to the appropriate width of an int.

We're just moving bytes! Once you understand what is happening "under the hood," there are multiple ways to do it (in some cases!)
# Pointers to Arrays — Memory Footprint

Full example:

```c
// file: pointer_to_array1.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

void swapA(int *arr, int index_x, int index_y)
{
    int tmp = *(arr + index_x);
    *(arr + index_x) = *(arr + index_y);
    *(arr + index_y) = tmp;
}

void swapB(int *arr, int index_x, int index_y)
{
    int tmp;
    memmove(&tmp, arr + index_x, sizeof(int));
    memmove(arr + index_x, arr + index_y, sizeof(int));
    memmove(arr + index_y, &tmp, sizeof(int));
}

void swapC(int *arr, int index_x, int index_y)
{
    char tmp[sizeof(int)];
    memmove(tmp, arr + index_x, sizeof(int));
    memmove(arr + index_x, arr + index_y, sizeof(int));
    memmove(arr + index_y, tmp, sizeof(int));
}

int main(int argc, char **argv)
{
    int values[] = {8, 2, 7, 14, -5, 42};
    swapA(values, 0, 5); // swaps 8 and 42
    swapB(values, 1, 2); // swaps 2 and 7
    swapC(values, 3, 4); // swaps 14 and -5
    int nelems = sizeof(arr) / sizeof(arr[0]);
    for (int i = 0; i < nelems; i++) {
        printf("%d", values[i]);
        i == nelems - 1 ? printf("\n") : printf(", ");
    }
    return 0;
}
```

```
$ ./pointer_to_array1
42, 7, 2, -5, 14, 8
```
For our second example, let's look at `argv`, which is an array of `char *` pointers:

```
./swapwords apple banana orange peach pear
```

Can we write a function to swap two pointers in the array?
For our second example, let's look at `argv`, which is an array of `char *` pointers:

```bash
./swapwords apple banana orange peach pear
```

Can we write a function to swap two pointers in the array? Sure:

```c
void swapA(char **arr, int index_x, int index_y) {
    char *tmp = *(arr + index_x);
    *(arr + index_x) = *(arr + index_y);
    *(arr + index_y) = tmp;
}
```

Note (very important!) -- Only the pointers are getting swapped! We are not copying the text from each string, at all. Just think of how we copy bytes.
For our second example, let's look at `argv`, which is an array of `char *` pointers:

```
./swapwords apple banana orange peach pear
```

Can we write a function to swap two pointers in the array **using memmove**?
For our second example, let's look at `argv`, which is an array of `char *` pointers:

```
./swapwords apple banana orange peach pear
```

Can we write a function to swap two pointers in the array **using memmove**?

```c
void swapB(char **arr, int index_x, int index_y)
{
    char *tmp;
    memmove(&tmp, arr + index_x, sizeof(char *));
    memmove(arr + index_x, arr + index_y,sizeof(char *));
    memmove(arr + index_y, &tmp,sizeof(char *));
}
```

In this case, we need to move 8 bytes at a time, and we conveniently get that value using `sizeof()`.
For our second example, let's look at `argv`, which is an array of `char *` pointers:

`./swapwords apple banana orange peach pear`

Once again...do we need to make `tmp` a `char *`?

```c
void swapB(char **arr, int index_x, int index_y)
{
    char *tmp;
    memmove(&tmp, arr + index_x, sizeof(char *));
    memmove(arr + index_x, arr + index_y, sizeof(char *));
    memmove(arr + index_y, &tmp, sizeof(char *));
}
```
Pointers to Arrays — Memory Footprint

For our second example, let's look at `argv`, which is an array of `char *` pointers:

```
./swapwords apple banana orange peach pear
```

Once again...do we need to make `tmp` a `char *`? **We can make it an array.**

```c
void swapC(char **arr, int index_x, int index_y)
{
    char tmp[sizeof(char *)];
    memmove(tmp, arr + index_x, sizeof(char *));
    memmove(arr + index_x, arr + index_y, sizeof(char *));
    memmove(arr + index_y, tmp, sizeof(char *));
}
```

It is a `char` array, because `char` is the only 1-byte type we have. Also note that we've removed the `&` from `tmp` in `memmove`, because it is now an array, which decays to a pointer when you use it as a parameter.
Full example:

```c
// file: swapwords.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

void swapA(char **arr, int index_x, int index_y)
{
    char *tmp = *(arr + index_x);
    *(arr + index_x) = *(arr + index_y);
    *(arr + index_y) = tmp;
}

void swapB(char **arr, int index_x, int index_y)
{
    char *tmp;
    memmove(&tmp, arr + index_x, sizeof(char *));
    memmove(arr + index_x, arr + index_y, sizeof(char *));
    memmove(arr + index_y, &tmp, sizeof(char *));
}

void swapC(char **arr, int index_x, int index_y)
{
    char tmp[sizeof(char *)];
    memmove(tmp, arr + index_x, sizeof(char *));
    memmove(arr + index_x, arr + index_y, sizeof(char *));
    memmove(arr + index_y, tmp, sizeof(char *));
}

int main(int argc, char **argv)
{
    if (argc < 6) {
        printf("Usage:\n\t%s s1 s2 s3 s4 s5\n", argv[0]);
        return -1;
    }
    // assume:
    // ./swapwords apple banana orange peach pear
    swapA(argv,1,5); // swaps apple and pear
    swapB(argv,2,3); // swaps banana and orange
    swapC(argv,3,4); // swaps banana and peach
    for (int i=1; i < argc; i++) { // skip programe
        printf("%s", argv[i]);
        i == argc - 1 ? printf("\n") : printf(", ");
    }
    return 0;
}
```

$ ./swapwords apple banana orange peach pear
pear, orange, peach, banana, apple
You might be asking -- why would we ever want to use the `memmove` function, if we already know the type?

Ah -- this is a key insight that we will discuss soon! When we get to "void *" pointers, we will find out that there is no way to do this without `memmove`, and we will actually need information about the width of the type itself!

Preview:

```c
void swap_generic(void *arr, int index_x, int index_y, int width)
{
    char tmp[width];
    void *x_loc = (char *)arr + index_x * width;
    void *y_loc = (char *)arr + index_y * width;

    memmove(tmp, x_loc, width);
    memmove(x_loc, y_loc, width);
    memmove(y_loc, tmp, width);
}
```
In CS 107, we are going to talk about two different areas of memory that your program will access, called the *stack* and the *heap*.

This diagram shows the overall memory layout in Linux on an x86-64 computer (e.g., the Myth computers).

Every program, by default, has access to an 8MB stack segment in memory. Your program can do anything it wants with that memory, but it is limited. The stack grows *downward* in memory, so your program starts with a location on the stack, and you get the next 8MB lower in memory.
Below the stack is the shared library. This is all of the standard libraries that are used by programs (e.g., stdlib.h, stdio.h, string.h, etc.) Your programs do not have access to these directly, except to call functions that are there.

Below the shared library data is the heap, which is managed by the operating system, and comprises the vast majority of the memory in your computer. When a program wants to use heap memory, it requests it from the operating system (using malloc, calloc, or realloc in C).

The heap starts at a low memory address and grows upwards.
Below the heap is global data for your program (i.e., global variables and string literals -- remember that string literals are not modifiable).

Below the global data is your program code.

**Note**: When you program references memory, it references *virtual* memory. Virtual memory is a way for every program to *think* it has access to the entire memory system, while hiding the details. The operating system and PC hardware handle all of the details of the translation between virtual memory and physical memory, and for this course you only need to consider the diagram to the left.
When a function creates a local variable, or when a function receives parameters, the data is either kept in *registers* or kept on the stack. We will cover registers when we get to assembly language, but for now we will assume that all of our local variables go on the stack (and we will compile with "-O0" which forces everything onto the stack.

Arrays are also kept on the stack.
Let's look at an example:

```c
// file: stack_ex1.c
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char **argv)
{
    int a = 0x12345;
    int b = 0x98765432;
    char str[] = "hello";
    short array[] = {0x2,0x4,0x6,0x8,0xa};

    printf("0x%x\n",a);
    printf("0x%x\n",b);
    printf("%s\n",str);
    for (int i=0; i < sizeof(array) / sizeof(array[0]); i++) {
        printf("0x%x,",array[i]);
    }
    printf("\n");
    return 0;
}
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffffffde984</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffde980</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffde97c</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffde978</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffde974</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffde970</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffde96c</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffde968</td>
<td></td>
</tr>
</tbody>
</table>
Stack Allocation

Let's look at an example:

```c
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char **argv) {
    int a = 0x12345;
    int b = 0x98765432;
    char str[] = "hello";
    short array[] = {0x2,0x4,0x6,0x8,0xa};

    printf("0x%x\n",a);
    printf("0x%x\n",b);
    printf("%s\n",str);
    for (int i=0; i < sizeof(array) / sizeof(array[0]); i++) {
        printf("0x%x,",array[i]);
    }
    printf("\n");
    return 0;
}
```

```
$ gcc -g -O0 -std=gnu99 -Wall stack_ex1.c -o stack_ex1
$ ./stack_ex1
0x12345
0x98765432
hello
0x2,0x4,0x6,0x8,0xa,
```

```
$ gdb stack_ex1
(gdb) break 12
Breakpoint 1 at 0x40067d: file stack_ex1.c, line 12.
(gdb) run
Starting program: stack_ex1
Breakpoint 1, main (argc=1, argv=0x7ffffffea78) at stack_ex1.c:12
12     printf("0x%x\n",a);
(gdb)
```

```
(gdb) p &a
$16 = (int *) 0x7fffffffde968
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffffffde984</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffde980</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffde97c</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffde978</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffde974</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffde970</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffde96c</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffde968</td>
<td></td>
</tr>
</tbody>
</table>
Let's look at an example:

```c
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char **argv) {
    int a = 0x12345;
    int b = 0x98765432;
    char str[] = "hello";
    short array[] = {0x2,0x4,0x6,0x8,0xa};

    printf("0x%x\n",a);
    printf("0x%x\n",b);
    printf("%s\n",str);
    for (int i=0; i < sizeof(array) / sizeof(array[0]); i++) {
        printf("0x%x," ,array[i]);
    }
    printf("\n");
    return 0;
}
```

$ gcc -g -O0 -std=gnu99 -Wall stack_ex1.c -o stack_ex1
$ ./stack_ex1
```
0x12345
0x98765432
hello
0x2,0x4,0x6,0x8,0xa,
```
$ gdb stack_ex1
(gdb) break 12
Breakpoint 1 at 0x40067d: file stack_ex1.c, line 12.
(gdb) run
Starting program: stack_ex1
Breakpoint 1, main (argc=1, argv=0x7ffffffea78) at stack_ex1.c:12
12   printf("0x%x\n",a);
(gdb)
(gdb) p &a
$16 = (int *) 0x7fffffffde984
```
0x12345
0x98765432
hello
0x2,0x4,0x6,0x8,0xa,
```
$ gdb stack_ex1
(gdb) break 12
Breakpoint 1 at 0x40067d: file stack_ex1.c, line 12.
(gdb) run
Starting program: stack_ex1
Breakpoint 1, main (argc=1, argv=0x7ffffffea78) at stack_ex1.c:12
12   printf("0x%x\n",a);
(gdb)
```
0x12345
0x98765432
hello
0x2,0x4,0x6,0x8,0xa,
```
$ gdb stack_ex1
(gdb) break 12
Breakpoint 1 at 0x40067d: file stack_ex1.c, line 12.
(gdb) run
Starting program: stack_ex1
Breakpoint 1, main (argc=1, argv=0x7ffffffea78) at stack_ex1.c:12
12   printf("0x%x\n",a);
(gdb)
```
0x12345
0x98765432
hello
0x2,0x4,0x6,0x8,0xa,
Stack Allocation

Let's look at an example:

```c
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char **argv) {
    int a = 0x12345;
    int b = 0x98765432;
    char str[] = "hello";
    short array[] = {0x2,0x4,0x6,0x8,0xa};

    printf("0x%x",a);
    printf("0x%x",b);
    printf("%s",str);
    for (int i=0; i < sizeof(array) / sizeof(array[0]); i++) {
        printf("0x%x",array[i]);
    }
    printf("\n");
    return 0;
}
```

$ gcc -g -o0 -std=gnu99 -Wall stack_ex1.c -o stack_ex1
$ ./stack_ex1
0x12345
0x98765432
hello
0x2,0x4,0x6,0x8,0xa,
$ gdb stack_ex1
(gdb) break 12
Breakpoint 1 at 0x40067d: file stack_ex1.c, line 12.
(gdb) run
Starting program: stack_ex1
Breakpoint 1, main (argc=1, argv=0x7fffffffe928) at stack_ex1.c:12
12     printf("0x%x
",a);
```

(gdb) p &a
$16 = (int *) 0x7fffffffde984
(gdb) p &b
$17 = (int *) 0x7fffffffde980
```

Address          Value
0x7fffffffde984
0x7fffffffde980
0x7fffffffde97c
0x7fffffffde978
0x7fffffffde974
0x7fffffffde970
0x7fffffffde96c
0x7fffffffde968
0x12345
Stack Allocation

Let's look at an example:

```c
// file: stack_ex1.c
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char **argv)
{
    int a = 0x12345;
    int b = 0x98765432;
    char str[] = "hello";
    short array[] = {0x2,0x4,0x6,0x8,0xa};

    printf("0x%x\n",a);
    printf("0x%x\n",b);
    printf("%s\n",str);
    for (int i=0; i < sizeof(array) / sizeof(array[0]); i++) {
        printf("0x%x,",array[i]);
    }
    printf("\n");
    return 0;
}
```

$ gcc -g -O0 -std=gnu99 -Wall
    stack_ex1.c -o stack_ex1
$ ./stack_ex1

Breakpoint 1 at 0x40067d: file
    stack_ex1.c, line 12.
    (gdb) run
Starting program: stack_ex1

Breakpoint 1, main (argc=1,
    argv=0x7ffffffea78) at stack_ex1.c:12
  12     printf("0x%x\n",a);
    (gdb)

(gdb) p &a
$16 = (int *) 0x7fffffffde968
(gdb) p &b
$17 = (int *) 0x7fffffffde96c

Address | Value
--------|------
0x7fffffffde984 | 
0x7fffffffde980 | 
0x7fffffffde97c | 
0x7fffffffde978 | 
0x7fffffffde974 | 
0x7fffffffde970 | 
0x7fffffffde96c | 
0x7fffffffde968 | 
0x98765432 | 
0x12345 |
Let’s look at an example:

```c
// file: stack_ex1.c
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char **argv) {
    int a = 0x12345;
    int b = 0x98765432;
    char str[] = "hello";
    short array[] = {0x2,0x4,0x6,0x8,0xa};
    printf("0x%x
",a);
    printf("0x%x
",b);
    printf("%s
",str);
    for (int i=0; i < sizeof(array) / sizeof(array[0]); i++) {
        printf("0x%x," ,array[i]);
    }
    printf("\n");
    return 0;
}
```

$ gcc -g -O0 -std=gnu99 -Wall stack_ex1.c -o stack_ex1
$ ./stack_ex1
0x12345
0x98765432
hello
0x2,0x4,0x6,0x8,0xa,
$ gdb stack_ex1
(gdb) break 12
Breakpoint 1 at 0x40067d: file stack_ex1.c, line 12.
(gdb) run
Starting program: stack_ex1
Breakpoint 1, main (argc=1, argv=0x7fffffffea78) at stack_ex1.c:12
12     printf("0x%x
",a);
(gdb)
(gdb) p &a
$16 = (int *) 0x7fffffffde968
(gdb) p &b
$17 = (int *) 0x7fffffffde96c
(gdb) p &array[0]
$18 = (short *) 0x7fffffffde970
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0x7fffffffde984</td>
<td>0x98765432</td>
</tr>
<tr>
<td>$0x7fffffffde980</td>
<td>0x12345</td>
</tr>
<tr>
<td>$0x7fffffffde97c</td>
<td>0x2</td>
</tr>
<tr>
<td>$0x7fffffffde978</td>
<td>0x4</td>
</tr>
<tr>
<td>$0x7fffffffde974</td>
<td>0x6</td>
</tr>
<tr>
<td>$0x7fffffffde970</td>
<td>0x8</td>
</tr>
<tr>
<td>$0x7fffffffde96c</td>
<td>0xa</td>
</tr>
</tbody>
</table>
Let's look at an example:

```c
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char **argv)
{
    int a = 0x12345;
    int b = 0x98765432;
    char str[] = "hello";
    short array[] = {0x2,0x4,0x6,0x8,0xa};
    printf("0x%x\n",a);
    printf("0x%x\n",b);
    printf("%s\n",str);
    for (int i=0; i < sizeof(array) / sizeof(array[0]); i++) {
        printf("0x%x,",array[i]);
    }
    printf("\n");
    return 0;
}
```

$ gcc -g -O0 -std=gnu99 -Wall
    stack_ex1.c -o stack_ex1
$ ./stack_ex1
0x12345
0x98765432
hello
0x2,0x4,0x6,0x8,0xa,
$ gdb stack_ex1
(gdb) break 12
Breakpoint 1 at 0x40067d: file stack_ex1.c, line 12.
(gdb) run
Starting program: stack_ex1
Breakpoint 1, main (argc=1, argv=0x7fffffffea78) at stack_ex1.c:12
12     printf("0x%x\n",a);
   12     printf("0x%x\n",a);
(gdb) p &a
$16 = (int *) 0x7fffffffe968
(gdb) p &b
$17 = (int *) 0x7fffffffe97c
(gdb) p &array[0]
$18 = (short *) 0x7fffffffe970
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xfffffffffee984</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffe980</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffe97c</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffe978</td>
<td>0xa</td>
</tr>
<tr>
<td>0x7fffffffe974</td>
<td>0x8</td>
</tr>
<tr>
<td>0x7fffffffe970</td>
<td>0x6</td>
</tr>
<tr>
<td>0x7fffffffe970</td>
<td>0x4</td>
</tr>
<tr>
<td>0x7fffffffe970</td>
<td>0x2</td>
</tr>
<tr>
<td>0x7fffffffe96c</td>
<td>0x98765432</td>
</tr>
<tr>
<td>0x7fffffffe968</td>
<td>0x12345</td>
</tr>
</tbody>
</table>
Let's look at an example:

```c
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char **argv) {
    int a = 0x12345;
    int b = 0x98765432;
    char str[] = "hello";
    short array[] = {0x2,0x4,0x6,0x8,0xa};
    printf("0x%x\n",a);
    printf("0x%x\n",b);
    printf("%s\n",str);
    for (int i=0; i < sizeof(array) / sizeof(array[0]); i++) {
        printf("0x%x,"array[i]);
    }
    printf("\n");
    return 0;
}
```

Compilation:

```
$ gcc -g -O0 -std=gnu99 -Wall stack_ex1.c -o stack_ex1
$ ./stack_ex1
```

Debugging:

```
(gdb) break 12
(gdb) run
```

```
Breakpoint 1, main (argc=1, argv=0x7fffffffea78) at stack_ex1.c:12
12     printf("0x%x\n",a);
(gdb)
```

```
(gdb) p &a
$16 = (int *) 0x7fffffffe968
(gdb) p &b
$17 = (int *) 0x7fffffffe96c
(gdb) p &array[0]
$18 = (short *) 0x7fffffffe970
(gdb) p &str[0]
$19 = 0x7fffffffe980 "hello"
```

Address | Value
--- | ---
0x7fffffffe984 | 
0x7fffffffe980 | 
0x7fffffffe974 | 0xa
0x7fffffffe970 | 0x8
0x7fffffffe96c | 0x6
0x7fffffffe968 | 0x4
0x98765432 | 0x2
0x12345 | 
0x7fffffffe978 | 0x98765432
Stack Allocation

Let's look at an example:

```c
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char **argv)
{
    int a = 0x12345;
    int b = 0x98765432;
    char str[] = "hello";
    short array[] = {0x2, 0x4, 0x6, 0x8, 0xa};
    printf("0x%x\n", a);
    printf("0x%x\n", b);
    printf("%s\n", str);
    for (int i=0; i < sizeof(array) / sizeof(array[0]); i++) {
        printf("0x%x,\n", array[i]);
    }
    printf("\n");
    return 0;
}
```

$ gcc -g -O0 -std=gnu99 -Wall stack_ex1.c -o stack_ex1
$ ./stack_ex1
0x12345
0x98765432
"hello"
0x2,0x4,0x6,0x8,0xa,

$ gdb stack_ex1
(gdb) break 12
Breakpoint 1 at 0x40067d: file stack_ex1.c, line 12.
(gdb) run
Starting program: stack_ex1
Breakpoint 1, main (argc=1, argv=0x7ffffffea78) at stack_ex1.c:12
12     printf("0x%x\n", a);
      ^
(gdb)
```

(gdb) p &a
$16 = (int *) 0x7fffffffde968
(gdb) p &b
$17 = (int *) 0x7fffffffde96c
(gdb) p &array[0]
$18 = (short *) 0x7fffffffde970
(gdb) p &str[0]
$19 = 0x7fffffffde980 "hello"
```

$ gcc -g -O0 -std=gnu99 -Wall stack_ex1.c -o stack_ex1
$ ./stack_ex1
0x12345
0x98765432
"hello"
0x2,0x4,0x6,0x8,0xa,

$ gdb stack_ex1
(gdb) break 12
Breakpoint 1 at 0x40067d: file stack_ex1.c, line 12.
(gdb) run
Starting program: stack_ex1
Breakpoint 1, main (argc=1, argv=0x7fffffffdeea78) at stack_ex1.c:12
12     printf("0x%x\n", a);
      ^
(gdb)
```

(gdb) p &a
$16 = (int *) 0x7fffffffde968
(gdb) p &b
$17 = (int *) 0x7fffffffde96c
(gdb) p &array[0]
$18 = (short *) 0x7fffffffde970
(gdb) p &str[0]
$19 = 0x7fffffffde980 "hello"
```

Stack Allocation

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffffffde984</td>
<td>0x10</td>
</tr>
<tr>
<td>0x7fffffffde980</td>
<td>0x8</td>
</tr>
<tr>
<td>0x7fffffffde974</td>
<td>0x6</td>
</tr>
<tr>
<td>0x7fffffffde970</td>
<td>0x4</td>
</tr>
<tr>
<td>0x7fffffffde96c</td>
<td>0x2</td>
</tr>
<tr>
<td>0x7fffffffde968</td>
<td>0x12345</td>
</tr>
<tr>
<td>0x98765432</td>
<td>0x12345</td>
</tr>
</tbody>
</table>

// file: stack_ex1.c
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char **argv)
{
    int a = 0x12345;
    int b = 0x98765432;
    char str[] = "hello";
    short array[] = {0x2, 0x4, 0x6, 0x8, 0xa};
    printf("0x%x\n", a);
    printf("0x%x\n", b);
    printf("%s\n", str);
    for (int i=0; i < sizeof(array) / sizeof(array[0]); i++) {
        printf("0x%x,\n", array[i]);
    }
    printf("\n");
    return 0;
}
Let's look at an example:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    int a = 0x12345;
    int b = 0x98765432;
    char str[] = "hello";
    short array[] = {0x2, 0x4, 0x6, 0x8, 0xa};

    printf("0x%lx\n", a);
    printf("0x%lx\n", b);
    printf("%s\n", str);
    for (int i=0; i < sizeof(array) / sizeof(array[0]); i++) {
        printf("0x%hx,", array[i]);
    }
    printf("\n");
    return 0;
}
```

$ gcc -g -O0 -std=gnu99 -Wall stack_ex1.c -o stack_ex1
$ ./stack_ex1
0x12345
0x98765432
hello
0x2, 0x4, 0x6, 0x8, 0xa,

(gdb)
```
(gdb) p &a
$16 = (int *) 0x7fffffffde968
(gdb) p &b
$17 = (int *) 0x7fffffffde96c
(gdb) p &array[0]
$18 = (short *) 0x7fffffffde970
(gdb) p &str[0]
$19 = 0x7fffffffde980 "hello"
(gdb) x/30bx &a
0x7fffffffde968:  0x45  0x23  0x01  0x00  0x32  0x54  0x76  0x98
0x7fffffffde970:  0x02  0x00  0x04  0x00  0x06  0x00  0x08  0x00
0x7fffffffde978:  0x0a  0x00  0x04  0x00  0x00  0x00  0x00  0x00
0x7fffffffde980:  0x68  0x65  0x6c  0x6c  0x6f  0x00
(gdb)
```

$ gdb stack_ex1
(gdb) break 12
Breakpoint 1 at 0x40067d: file stack_ex1.c, line 12.
(gdb) run
Starting program: stack_ex1
Breakpoint 1, main (argc=1, argv=0x7ffffffea78) at stack_ex1.c:12
12     printf("0x%lx\n", a);
   12     printf("0x%lx\n", b);
   12     printf("%s\n", str);
   12     for (int i=0; i < sizeof(array) / sizeof(array[0]); i++) {
   12         printf("0x%hx,", array[i]);
   12     }
   12     printf("\n");
   12     return 0;
   12
(gdb) p &a
$16 = (int *) 0x7fffffffde968
(gdb) p &b
$17 = (int *) 0x7fffffffde96c
(gdb) p &array[0]
$18 = (short *) 0x7fffffffde970
(gdb) p &str[0]
$19 = 0x7fffffffde980 "hello"
(gdb) x/30bx &a
0x7fffffffde968:  0x45  0x23  0x01  0x00  0x32  0x54  0x76  0x98
0x7fffffffde970:  0x02  0x00  0x04  0x00  0x06  0x00  0x08  0x00
0x7fffffffde978:  0x0a  0x00  0x04  0x00  0x00  0x00  0x00  0x00
0x7fffffffde980:  0x68  0x65  0x6c  0x6c  0x6f  0x00
(gdb)
Every time you call a function, the data for the current function needs to be saved. The x86 operating system handles this in an elegant manner: a function just uses memory farther down in the stack, and leaves the part of the stack that the calling function was using unchanged. Example:

```c
int rotate_rgb(int rgb)
{
    // rgb -> brg
    int g = ((rgb & 0xff) << 16);
    return (rgb >> 8) | g;
}

void colorize(int *colors, size_t nelems)
{
    for (int i=0; i < nelems; i++) {
        colors[i] = rotate_rgb(colors[i]);
    }
}

int main(int argc, char **argv)
{
    size_t nelems = argc-1;
    int values[nelems];
    char *err;
    for (int i=0; i < argc-1; i++) {
        values[i] = strtol(argv[i+1],&err,0);
    }
    print_colors(values, nelems);
    colorize(values,sizeof(values)/sizeof(values[0]));
    print_colors(values, nelems);
    return 0;
}
```
Every time you call a function, the data for the current function needs to be saved. The x86 operating system handles this in an elegant manner: a function just uses memory farther down in the stack, and leaves the part of the stack that the calling function was using unchanged. Example:

```c
int rotate_rgb(int rgb)
{
    // rgb -> bgr
    int g = ((rgb & 0xff) << 16);
    return (rgb >> 8) | g;
}

void colorize(int *colors, size_t nelems)
{
    for (int i=0; i < nelems; i++) {
        colors[i] = rotate_rgb(colors[i]);
    }
}

int main(int argc, char **argv)
{
    size_t nelems = argc-1;
    int values[nelems];
    char *err;
    for (int i=0; i < argc-1; i++) {
        values[i] = strtol(argv[i+1],&err,0);
    }
    print_colors(values, nelems);
    colorize(values, sizeof(values)/sizeof(values[0]));
    print_colors(values, nelems);
    return 0;
}
```
Stack Frames

Every time you call a function, the data for the current function needs to be saved. The x86 operating system handles this in an elegant manner: a function just uses memory farther down in the stack, and leaves the part of the stack that the calling function was using unchanged. Example:

```c
int rotate_rgb(int rgb)
{
    // rgb -> brg
    int g = ((rgb & 0xff) << 16);
    return (rgb >> 8) | g;
}

void colorize(int *colors, size_t nelems)
{
    for (int i=0; i < nelems; i++) {
        colors[i] = rotate_rgb(colors[i]);
    }
}

int main(int argc, char **argv)
{
    size_t nelems = argc-1;
    int values[nelems];
    char *err;
    for (int i=0; i < argc-1; i++) {
        values[i] = strtol(argv[i+1],&err,0);
    }
    print_colors(values, nelems);
    colorize(values,sizeof(values)/sizeof(values[0]));
    print_colors(values, nelems);
    return 0;
}
```
Every time you call a function, the data for the current function needs to be saved. The x86 operating system handles this in an elegant manner: a function just uses memory farther down in the stack, and leaves the part of the stack that the calling function was using unchanged. Example:

```c
int rotate_rgb(int rgb)
{
    // rgb -> brg
    int g = ((rgb & 0xff) << 16);
    return (rgb >> 8) | g;
}

void colorize(int *colors, size_t nelems)
{
    for (int i=0; i < nelems; i++) {
        colors[i] = rotate_rgb(colors[i]);
    }
}

int main(int argc, char **argv)
{
    size_t nelems = argc-1;
    int values[nelems];
    char *err;
    for (int i=0; i < argc-1; i++) {
        values[i] = strtol(argv[i+1],&err,0);
    }
    print_colors(values, nelems);
    colorize(values,sizeof(values)/sizeof(values[0]));
    print_colors(values, nelems);
    return 0;
}
```
Stack Frames

Every time you call a function, the data for the current function needs to be saved. The x86 operating system handles this in an elegant manner: a function just uses memory farther down in the stack, and leaves the part of the stack that the calling function was using unchanged. Example:

```c
int rotate_rgb(int rgb)
{
    // rgb -> brg
    int g = ((rgb & 0xff) << 16);
    return (rgb >> 8) | g;
}

void colorize(int *colors, size_t nelems)
{
    for (int i=0; i < nelems; i++) {
        colors[i] = rotate_rgb(colors[i]);
    }
}

int main(int argc, char **argv)
{
    size_t nelems = argc-1;
    int values[nelems];
    char *err;
    for (int i=0; i < argc-1; i++) {
        values[i] = strtol(argv[i+1],&err,0);
    }
    print_colors(values, nelems);
    colorize(values,sizeof(values)/sizeof(values[0]));
    print_colors(values, nelems);
    return 0;
}
```
Every time you call a function, the data for the current function needs to be saved. The x86 operating system handles this in an elegant manner: a function just uses memory farther down in the stack, and leaves the part of the stack that the calling function was using unchanged. Example:

```c
int rotate_rgb(int rgb)
{
    // rgb -> bgr
    int g = ((rgb & 0xff) << 16);
    return (rgb >> 8) | g;
}

void colorize(int *colors, size_t nelems)
{
    for (int i=0; i < nelems; i++) {
        colors[i] = rotate_rgb(colors[i]);
    }
}

int main(int argc, char **argv)
{
    size_t nelems = argc-1;
    int values[nelems];
    char *err;
    for (int i=0; i < argc-1; i++) {
        values[i] = strtol(argv[i+1],&err,0);
    }
    print_colors(values, nelems);
    colorize(values,sizeof(values)/sizeof(values[0]));
    print_colors(values, nelems);
    return 0;
}
```
Parameters can also be put onto the stack, and they just behave like local variables.

They might actually point to other elements on the stack.

In our example, colors points to the values array. Non-pointers are just copied (e.g., nelems).
Why we like stack allocation

It is fast. Allocating space on the stack is efficient because your program already has access to the memory.

It is convenient. When you leave a function, all your stack-allocated data is left in place, and there isn't anything to clean up. Think of the stack as "scratch space" where your program can jot things down when it needs them inside a function. The scope (lifetime) of the data is inside the function, so it keeps things tidy.

Type safety. You are controlling the type of the variables, and therefore the compiler can do lots of checks on the data. We will see that this isn't always the case with heap memory.
Why we dislike stack allocation

It isn't that plentiful. You're limited to 8MB of data for your program, by default (you can change this before you run the program if you want). This might seem like a good deal of space, but if your program needs more space, you can't get it from the stack!

Size fixed at declaration, with no option to resize. You can't resize an array, and once you allocate it, it is there for the lifetime of your function or block.

Limited scope. Once the function or block is finished, your stack-based memory is gone! You can't return a pointer to a stack array, for instance (well, you can, but your program will be corrupted).
3 minute break
"Dynamic allocation" should be familiar to you if you took CS 106B, where you used the `new` and `delete` operators to request memory for arrays and objects.

In C, we don't have objects, but we can request memory from the heap, using three functions:

- `malloc`
- `calloc`
- `realloc`

and we return the memory to the operating system using `free`. 
The most common method for requesting memory from the heap is by using `malloc`. The function is used to allocate a specified number of bytes:

```c
void *malloc(size_t size);
```

Size is always in **bytes**, so often you need to calculate the number of bytes with `sizeof` and a multiplication.

`malloc` returns a "`void *" pointer, which basically means that you can assign the return value to any pointer. Example:

```c
int *scores = malloc(20 * sizeof(int)); // allocate an array of 20 ints.
In reality, this is just an allocation of 80 bytes, which the compiler will treat as an int array)
```

If `malloc` returns `NULL`, then there wasn't enough memory for the request. :(
calloc is like malloc, except that it takes two parameters which are multiplied to calculate the number of bytes, and it **zeros** the memory for you (**malloc** does not zero the memory!*)

```c
void *calloc(size_t nmemb, size_t size);
```

`nmemb * size` will be bytes, so the following would be functionally equivalent:

```c
int *scores = calloc(20, sizeof(int)); // allocate and zero 20 ints

// alternate (but potentially slower)
int *scores = malloc(20 * sizeof(int)); // allocate an array of 20 ints.
for (int i=0; i < 20; i++) scores[i] = 0;
```

* it's a bit more subtle than that -- new memory that your process hasn't used before will be zeroed for security reasons by **malloc**, but if the OS re-issues you memory, it won't be zeroed.
realloc can be used to (potentially) change the size of the memory block pointed to by its pointer:

```c
void *realloc(void *ptr, size_t size);
```

The `realloc` function returns a pointer to the memory block, which will often be the same pointer you pass in as `ptr`. If it needs to move the data, it moves it for you, and then passes back a different pointer. If the request fails, it returns `NULL`, but the original memory is not affected (e.g., your original pointer is still valid).

Example:

```c
int *values = malloc(10 * sizeof(int)); // allocate space for 10 ints
... // fill up values, etc.
int *new_values = realloc(20 * sizeof(int)); // increase the memory to 20 ints
if (new_values != NULL) values = new_values;
else { ...request failed, deal with gracefully }
```
When a function uses `malloc`, `calloc`, and `realloc`, the function is responsible for returning the memory to the operating system when it no longer needs it. Un-returned memory is called a *memory leak*, and wastes memory.

To return memory, the `free` function is used:

```c
void free(void *ptr);
```

`ptr` must point to a previously allocated block (or it can be `NULL`). Once a program frees memory, *it cannot be used again*. The pointer can, of course, be re-used to point elsewhere.
We can use valgrind to determine if there are memory leaks:

$ valgrind ./allocation 90 85 92
90,85,92,90,95

==6038== Memcheck, a memory error detector
==6038== Copyright (C) 2002-2015, and GNU GPL’d, by Julian Seward et al.
==6038== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==6038== Command: ./allocation 90 85 92
==6038== 90,85,92,90,95
==6038== ==6038== HEAP SUMMARY:
==6038==     in use at exit: 0 bytes in 0 blocks
==6038==   total heap usage: 3 allocs, 3 frees, 1,056 bytes allocated
==6038== ==6038== All heap blocks were freed -- no leaks are possible
==6038== ==6038== For counts of detected and suppressed errors, rerun with: -v
==6038== ==6038== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

You want to see the "All heap blocks were freed" message.
• **References:**
  • K&R C Programming (from our course)
  • Course Reader, C Primer

• **Advanced Reading:**
  • virtual memory: [https://en.wikipedia.org/wiki/Virtual_memory](https://en.wikipedia.org/wiki/Virtual_memory)