Programming paradigms

Based on Stanford CS107 Programming paradigms

Gabriel Barberini 2024/01

I dedicate these notes to Helena, my newborn daughter

Procedural programming

Memory fundamentals

Recap on bits and bytes

A Bit is a contraction of the word *binary unit* and is a fundamental unit of information, it is binary in the sense that every bit represents two possible states (e.g 1 or 0)

- It is commonly used in digital systems because electronic circuits can easily distinguish between two states (on/off, high/low voltage)

- 1 bit can represent $2^1 = 2$ values e.g {0,1}
- 2 bits can represent $2^2 = 4$ values e.g {00, 01, 10, 11} or {0, 1, 2, 3}
- n bits can represent 2^n values e.g $\{000\ldots0_n, 100\ldots0_n, \ldots, 111\ldots1_n\}$ or $[0, 2^n[$

- The binary number system is also a positional number system, just like decimal, therefore:

$$101_{binary} = 0b101 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5 = 5 \cdot 10^0 = 0d5$$

$$123_{decimal} = 0d123 =$$

$$1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 =$$

$$0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + \ldots + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 =$$

$0b01111011$

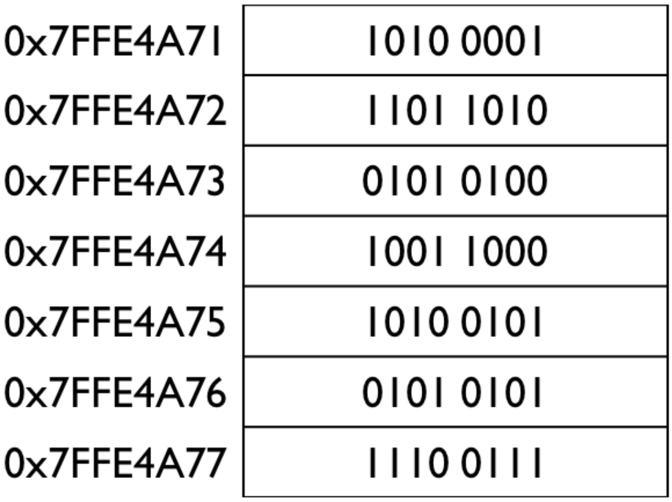
=> $(1, 2, 3)_{10} = (0, 1, 1, 1, 1, 0, 1, 1)_2$

- 1 byte is equal to 8 bits

Recap on memory addresses

- A 32 bit flat-memory model architecture is said to have 2^{32} distinct memory addresses, usually each can hold 1 byte of information → up to 2^{32} bytes of memory available
- A 64 bit architecture → up to 2^{64} bytes of memory available

- The addresses are usually shown in hexadecimal (i.e. 0x00000000 to 0xFFFFFFFF)



0x7FFE4A71	1010 0001
0x7FFE4A72	1101 1010
0x7FFE4A73	0101 0100
0x7FFE4A74	1001 1000
0x7FFE4A75	1010 0101
0x7FFE4A76	0101 0101
0x7FFE4A77	1110 0111

Further reading: https://elec2645.github.io/104/pointers.html

---

Meaning of a "word"

Generally refers to the natural data unit used by a particular processor design.

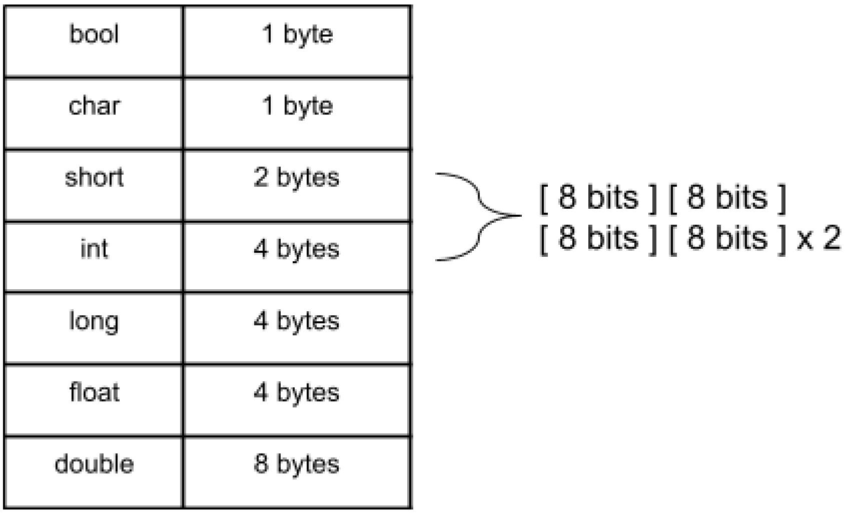
- **Half-Word**: Generally refers to half the size of the word.
- **Double-Word**: Generally refers to twice the size of the word.

Word Size

- **1 Byte (8 bits)**: A word could be as small as 1 byte on some early computers and microcontrollers. However, this is less common in modern computing.
- **2 Bytes (16 bits)**: Some older and embedded systems might use 16-bit words. This was common in 16-bit architectures.
- **4 Bytes (32 bits)**: On 32-bit architectures, a word is typically 4 bytes (32 bits).
- **8 Bytes (64 bits)**: On 64-bit architectures, a word is usually 8 bytes (64 bits).

The size of a word has an intrinsic relationship with the operation step at the register level within the assembly code. Note that for virtually all *modern* architectures: each unique memory address typically corresponds to a single byte of storage, even if the CPU is *fetching* or *operating* on a full word (e.g., 4 bytes at a time on a 32-bit system), the actual addressability is still at the byte level.

Basic data types



bool	1 byte
char	1 byte
short	2 bytes
int	4 bytes
long	4 bytes
float	4 bytes
double	8 bytes

[8 bits] [8 bits]
[8 bits] [8 bits] x 2

Signed magnitude

- Represents negative numbers with a leftmost 1 bit and positive numbers with a leftmost 0 bit

One's complement

- Represents negative numbers as the bit flip inverse of a positive integer

- The leftmost bit is used to determine the sign of the number, when it is 0 the number is positive and when it is 1 the number is negative

- It can map up to

$$\left[-\overbrace{2^{N-1}}^{\text{left-most bit discarded}}\underbrace{-\ 1,}_{\text{0 included}}\ 2^{N-1}-1\right]$$

non-zero integers

N: amount of bits available

e.g

- left-most bit 0 → (+)

0b	0d
0111 1110	126
0111 1111	127

- left-most bit 1 → (-)

0b	0d
1000 0001	-126
1000 0000	-127

Note that 0 can be represented in two ways

`short 0` in one's complement

0	00000000 00000000	11111111 11111111	-0

Two's complement

- Used in most systems
- Represents 0 without ambiguity

- Starts from one's complement and adds 1 bit **to the negative counterpart only**

e.g

`short 0` on `-0` form in one's complement

0	00000000 00000000	11111111 11111111 +	-0

		00000000 00000001	

overflows to …

=

`short 0` *in two's complement*

0	00000000 00000000	00000000 00000000	0

- Which ends up giving more space to represent one additional negative integer, therefore two's complement maps up to:

  $$[-\overbrace{2^{N-1},}^{\text{left-most bit discarded}}\ 2^{N-1}-1]$$

  non-zero integers, which is one more than one's complement

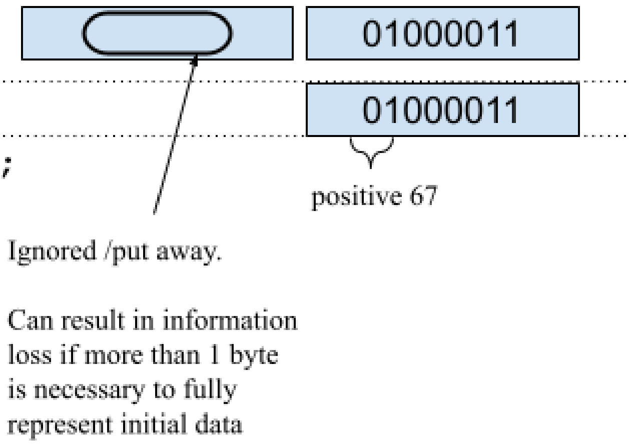
  N: amount of bits available

- Further reading on signed magnitude vs one's complement vs two's complement: https://www.electronics-tutorials.ws/binary/signed-binary-numbers.html

Type casting

Largest to smallest

```
short s = 67;
char ch = s;
cout << ch << endl;
> c
```



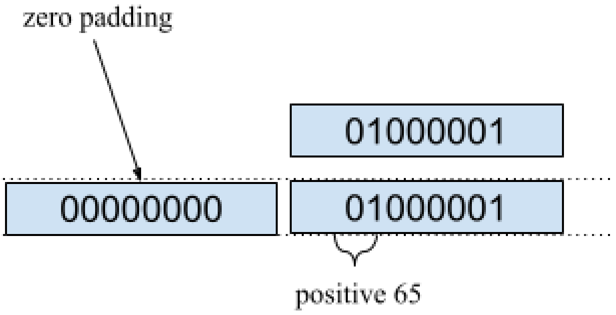
Ignored /put away.

Can result in information loss if more than 1 byte is necessary to fully represent initial data

---

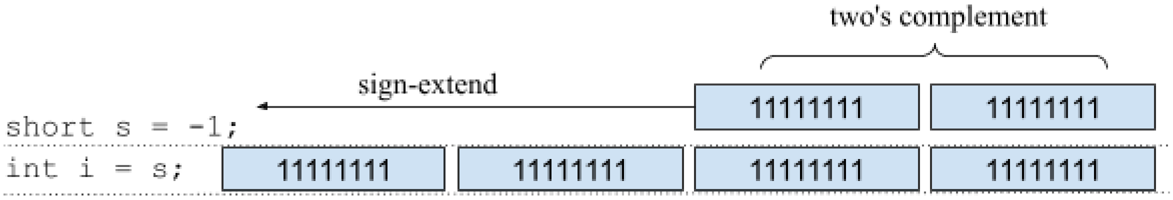
Smallest to largest

```
char ch = 'A';
short s = ch;
cout << s << endl;
> 01000001
```



Casting signal preservation

- Casting smallest to largest preserves signal through sign-extension

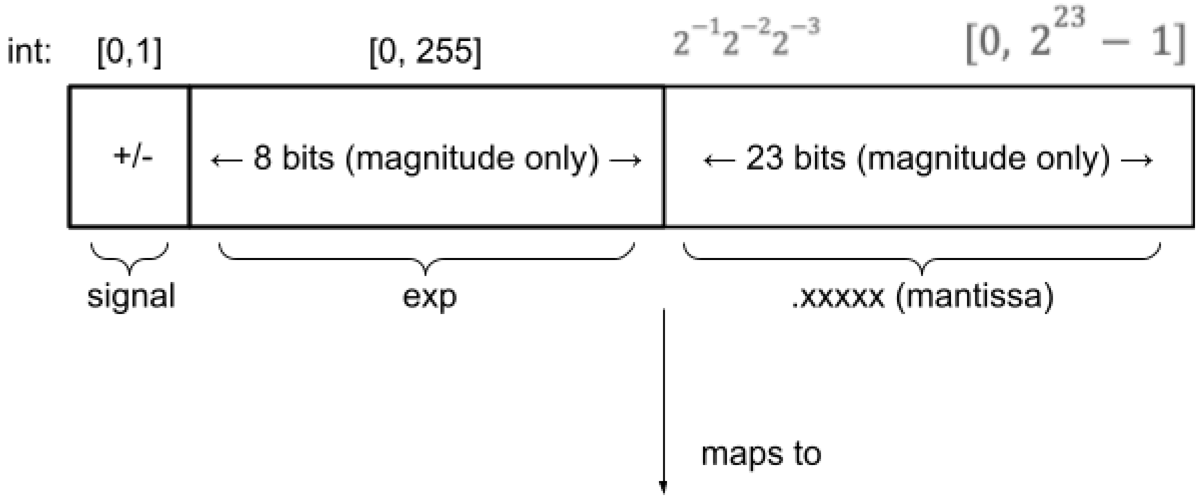


```
short s = -1;
int i = s;
```

So that 2'C(-1 + 1) = 0 = 00000000 00000000 00000000 00000000

Floats

Memory representation (IEE 754)



$$\boxed{-\ 1^{signal} \cdot 1.xxxxx \cdot 2^{exp-127}}$$

$$\boxed{\{1,\text{-}1\}\ \{1.\ [0,\ 2^{23} - 1]\ \}\{\ \ [2^{-127},\ 2^{-128}]\ \}}$$

e.g:

$$\begin{aligned} & 7.0 \\ \Leftrightarrow\ & 7.0\ \cdot 2^{0} \\ \Leftrightarrow\ & 3.5\ \cdot 2^{1} \\ \Leftrightarrow\ & 1.75\ \cdot 2^{2} \\ \Leftrightarrow\ & (-\ 1^{0})\ \cdot\ (1.75)\ \cdot\ (2^{2}) \end{aligned}$$

$\therefore$

→ signal = 0

→ .xxxxx = 75

→ exp = 129

Integer conversion

int → float

```
int i = 5;
float f = i;
cout << f << endl;
> 5.0
```

Machine operates in `int` to find **signal, exponent** and **mantissa** to calculate the IEE 754 float representation.

---

float → int

Machine calculates the effective `float` value according to **signal**, **exponent** and **mantissa** and then truncate or round the value. According to ISO/IEC 9899:2018 (C18 Standard) on float → int conversion C truncates discarding everything after the decimal point.

See https://cplusplus.com/forum/beginner/60827

Floating-point arithmetic

Floating-point arithmetic operations, such as addition and division, approximate the corresponding real number arithmetic operations by rounding any result that is not a floating-point number itself to a nearby floating-point number. For example, in floating-point arithmetic with five base-ten digits of precision, the sum 12.345 + 1.0001 = 13.3451 might be rounded to 13.345.

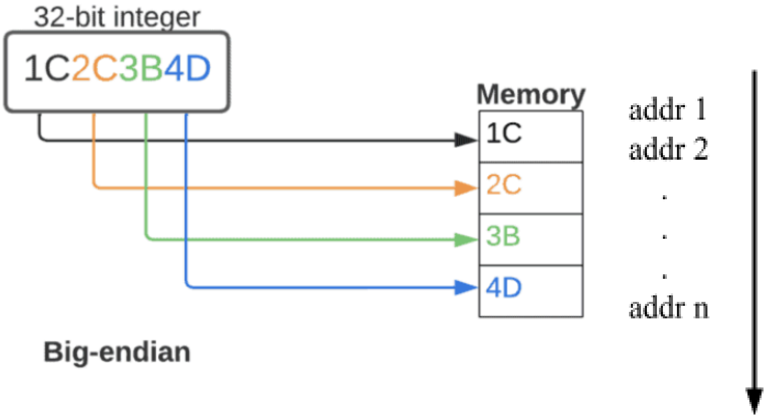
Pointers

Big endian vs little endian

Big-endian and little-endian are the two main ways to represent endianness.

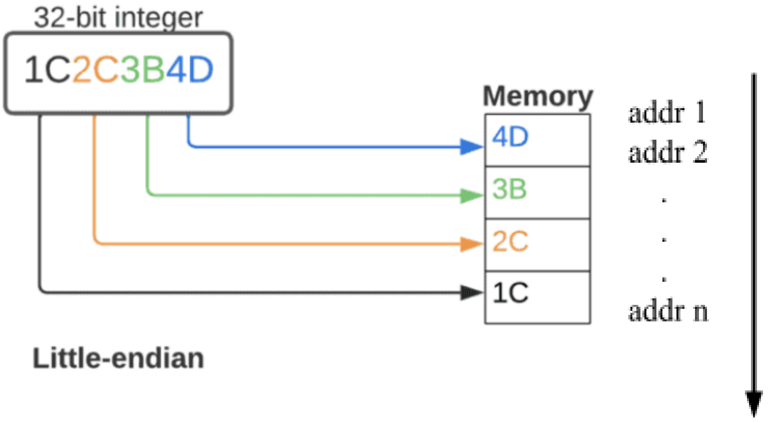
- Big-endian keeps the **most significant byte** of a word at the **smallest memory location** and the **least significant byte** at the **largest**.

```
left → right
```



- Little-endian keeps the **least significant** byte at the **smallest memory location**.

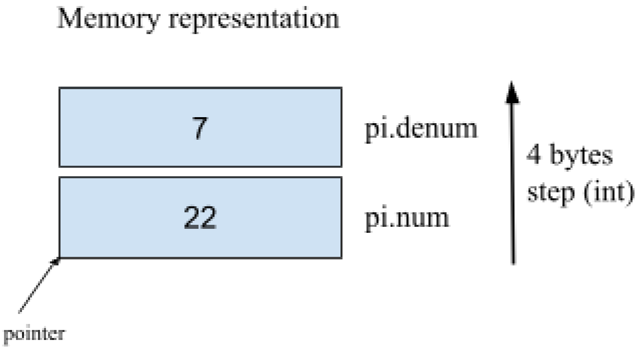
```
left ← right
```



Further reading: baeldung.com/cs/big-endian-vs-little-endian; en.wikipedia.org/wiki/Endianness

C structs

```
struct fraction {
    int num;
    int denum;
};

fraction pi;
pi.num = 22;
pi.denum = 7;
```



Order of memory allocation

- **Variable Allocation Order**: The order in which variables are allocated in memory can depend on the compiler and its optimizations. Generally, variables are allocated in the order they are declared (left→right; top→down), but this is not strictly guaranteed as the compiler may reorder for optimization purposes.

  e.g

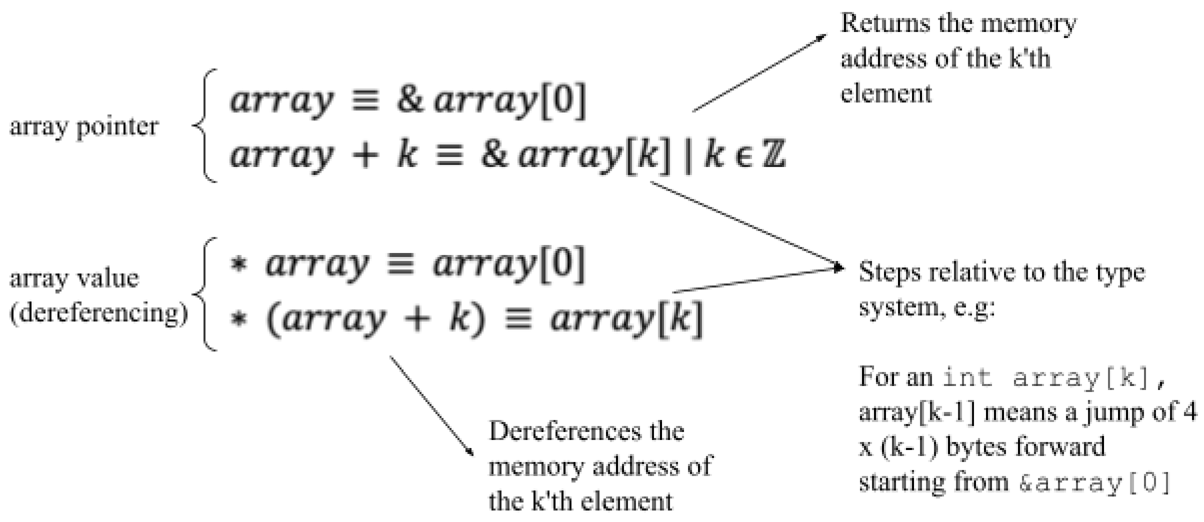
```
char sentence[] = "Hello";

| Memory Address  | Data  |
|-----------------|-------|
| ...             | ...   |
| 0x2000          | 'H'   |
| 0x2001          | 'e'   |
| 0x2002          | 'l'   |
| 0x2003          | 'l'   |
| 0x2004          | 'o'   |
| 0x2005          | '\0'  |
| ...             | ...   |
```

```
int x;
float y;

| Memory Address  | Data   |
|-----------------|--------|
| ...             | ...    |
| 0x1000          | x      |
| 0x1004          | y      |
| ...             | ...    |
```

- **Endianness**: Endianness (big-endian or little-endian) is determined by the machine architecture, not by the C code itself. But usually it is little-endian.

Pointer arithmetic

- Pointer is an object that stores a memory address. As an object, it has its own memory address as well.



array pointer

$$array \equiv \& array[0]$$

$$array + k \equiv \& array[k] \mid k \in \mathbb{Z}$$

array value (dereferencing)

$$* \, array \equiv array[0]$$

$$* \, (array + k) \equiv array[k]$$

Returns the memory address of the k'th element

Steps relative to the type system, e.g:

For an `int array[k]`, array[k-1] means a jump of 4 x (k-1) bytes forward starting from `&array[0]`

Dereferences the memory address of the k'th element

e.g

```
int ar[2];
ar[0] = 1
ar[1] = 2

printf("%d\n", ar[0])
> 1

printf("%d\n", ar[1])
> 2

printf("%p\n", &ar[0])
> 0x10e374000

printf("%p\n", &ar[1])
> 0x10e374004
```

```
printf("%p\n", &ar[1] + 1)
>  0x10e374008
```

When working on a 32-bit system of a flat memory model a pointer address costs 32 bits of information, hence 4 different memory addresses are needed to store the pointer address itself.

To clarify:

- **Pointer Size**: On a 32-bit system, the size of a pointer is 4 bytes (32 bits).

  - Each byte in memory has its own unique address.
  - The 4 bytes required to store a pointer are contiguous, meaning they occupy 4 consecutive memory addresses.
  - It only takes one pointer to reference any address within the 4 GB addressable memory space of a 32-bit flat-model system.

Further reading: https://stackoverflow.com/questions/20763616/how-many-bytes-do-pointers-take-up

---

Dereferencing

- Dereferencing is a technique for accessing or manipulating data stored in a memory location pointed to by a pointer. We use the * symbol with the pointer variable when dereferencing the pointer variable. Using dereferencing, we can get the value inside the variable.

- When you want to access the data/value in the memory that the pointer points to - the contents of the address with that numerical index - then you dereference the pointer.

Further reading:
https://stackoverflow.com/questions/4955198/what-does-dereferencing-a-pointer-mean-in-c-c

Dereferencing a null pointer

```
int c;
int *pi;
pi = NULL;
c = *pi;
```

- This operation will crash the application
- Never do this
- https://en.wikipedia.org/wiki/2024_CrowdStrike_incident

Further reading:
https://stackoverflow.com/questions/4007268/what-exactly-is-meant-by-de-referencing-a-null-pointer

---

Void pointer

A void pointer is a pointer that has no associated data type with it.

```
int a = 10;
void *p = &a;
```

- Can hold an address of any data type
- Can be cast into any data type
- Has an undefined type system and thus cannot be dereferenced

Void pointer arithmetic

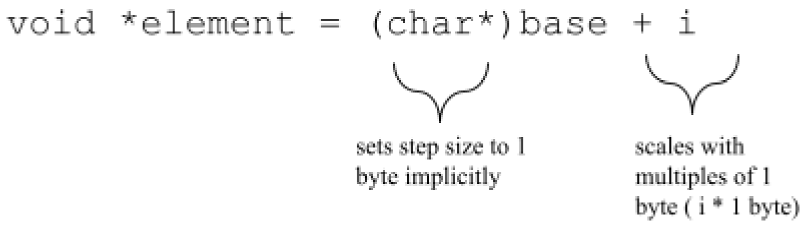
In C, pointer arithmetic is based on the size of the type that the pointer points to. When you perform an addition operation on a pointer, the compiler scales the integer by the size of the type that the pointer points to. Therefore, the compiler doesn't know how to handle the following code:

```
int i = 5;
void *base;
void *element = base + i;
```

Because behind the scenes, it would be trying to do `` `base + 5 * sizeof(void)` ``, and `` `void` `` has no predefined size. Although standard C does not allow arithmetic on void pointers, GNU C (GCC) does allow it and assumes the step size within a void pointer type system is 1 byte.

In an expression such as `` `*(array+i)` `` or `` `*(array-i)` `` the compiler is implicitly scaling `` `i` `` to the size of the type the array is deferred, so it also doesn't know what to do in an array of `` `void` ``.

But we can trick it to perform basic math and pointer arithmetic at the same time by using a 1:1 base for both domains (works the same for subtraction and addition):



```
void *element = (char*)base + i
```

sets step size to 1 byte implicitly

scales with multiples of 1 byte (i * 1 byte)

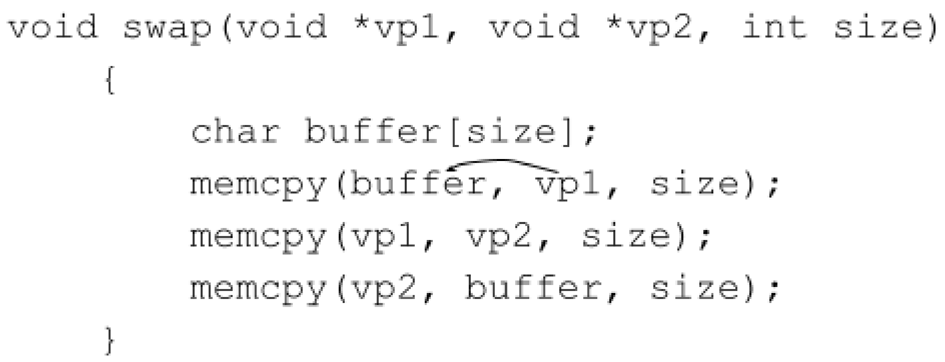
Generics

Motivational:

```
void swap(int *ap, int *bp)
    {
         int temp = *ap;
         *ap = *bp;
         *bp = temp;
    }
```

What if we wanted to swap content from `` `float` ``, `` `char` ``, `` `short` ``, or any other type-system pointer?

Well, for that we can use generics.



```
void swap(void *vp1, void *vp2, int size)
    {
        char buffer[size];
        memcpy(buffer, vp1, size);
        memcpy(vp1, vp2, size);
        memcpy(vp2, buffer, size);
    }
```

Obs: `memcpy` copies the values of num bytes from the location pointed to by source directly to the memory block pointed to by destination.

Advantage of using this approach over using templates:
- Single assembly for many data types → avoids inflating the binary with compiler-specific code

Disadvantage:
- Requires client to comply with code;
- Loose of safety (non-compliance of client can lead to hidden bugs)

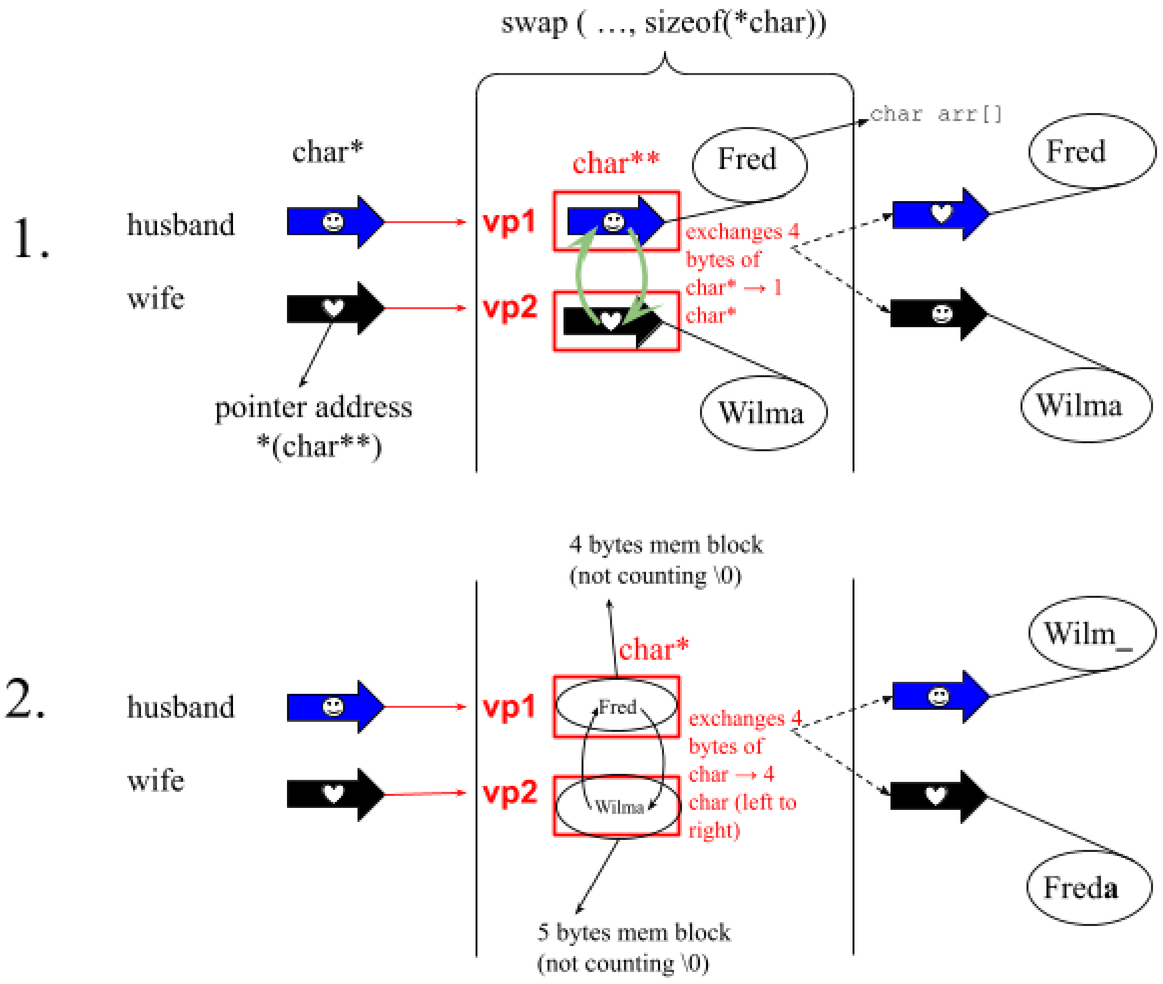
Swapping chars

Now suppose we have:

```
char *husband = strdup("Fred");
char *wife = strdup("Wilma");
```

And using the `swap` implementation above we perform:

```
1. swap(&husband, &wife, sizeof(char*));
2. swap(husband, wife, sizeof(char*));
```



obs: `sizeof(char*)` → 4 bytes in 32 bits flat-memory model

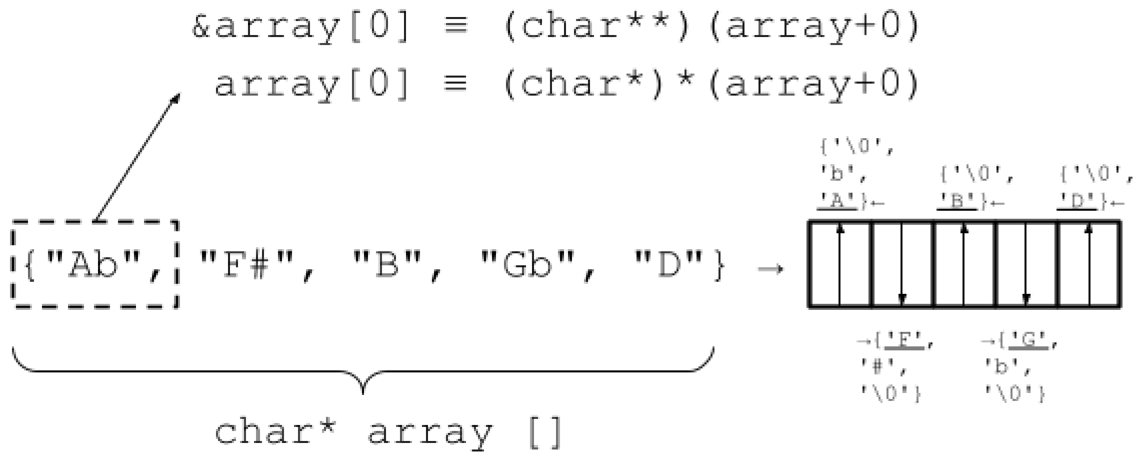
Pointers to pointers

There is no guarantee that `sizeof(n*) == sizeof(m*)` $\forall\, m, n \in \{type\ system\};$ The size of a pointer is architecture-specific, hence we should always emphasize memory address types.

Recap on C strings

- Each char in C represents one byte (8 bits) of information;
- A single memory address can hold 1 byte of information (on a byte-wide flat-memory model). Thus, each memory address can hold one char;
- **Strings are arrays of chars;**
  e.g: `char array[] = "abc"`
  which is equivalent to `char array[] = {'a', 'b', 'c', '\0'}`
  thus `sizeof(array) == strlen(array)+1`

Array of pointers to pointers

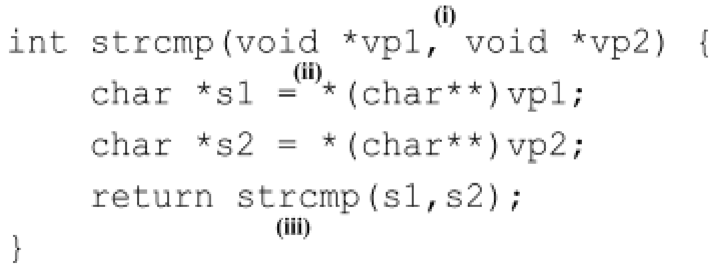

&array[0] ≡ (char**)(array+0)
array[0] ≡ (char*)*(array+0)
{"Ab", "F#", "B", "Gb", "D"} →
char* array []
{'\0',
'b',
'A'}←
{'\0',
'B'}←
{'\0',
'D'}←
→{'F',
'#',
'\0'}
→{'G',
'b',
'\0'}

informally:

```
char* array[] = {
                    →char array_1[], →char array_2[],
                    →char array_3[], …
                }
```

For most compilers both `char array[] = "AbF#BGbD"` and `char* array[] = {"Ab", "F#", "B", "Gb", "D"}` allocate memory contiguously from the first char to the last, the difference is that the `char* array[]` approach will have a null character `\0` for each substring while `char array[]` will have only one. Think of `char* array[]` as chopping a string into distinct and "pointable" slices (substrings) located one after the other in memory. But be careful, there is no reason for that to be always true, it just happens to be that way in most cases, for more information see https://stackoverflow.com/questions/51697188/in-array-of-pointers-to-string-will-all-the-strings-be-stored-in-contiguous-mem

Single dereferencing of a char **



```
int strcmp(void *vp1, void *vp2) {
    char *s1 = *(char**)vp1;
    char *s2 = *(char**)vp2;
    return strcmp(s1,s2);
}
```

(i) → Receives generic (void) pointer.

(ii) → Casts pointers to known type `` `char**` `` and dereferences it once, evaluating the amount of bytes equal the size of a pointer in the system architecture (4 bytes for 32 bits or 8 bytes for 64 in a flat-memory model), which in this case is an address assumed to be a `` `char*` `` pointer.

(iii) → Uses `` `strcmp` `` built-in function to compare each `` `char` `` inside `` `char array[]` ``. Here *strcmp* assumes input pointers are a `` `char*` `` address pointing to the first element of a string.

Recap on function pointers

Function Pointers provide an extremely interesting, efficient, and elegant programming technique. You can use them to replace switch/if-statements and to realize late-binding. Late binding is deciding the proper function during runtime instead of compiling time. They are less error-prone than normal pointers because you will never allocate or deallocate memory with them.

More info at:
https://www.cs.cmu.edu/~ab/15-123N09/lectures/Lecture%2008%20-%20Function%20Pointers.pdf

Functions like variables, can be associated with an address in the memory. We call this a function pointer. A specific function pointer variable can be defined as follows:

```
int (*fn)(int,int) ;
```

Here we define a function pointer `` `fn` ``, which can be initialized to any function that takes two integer arguments and returns an integer. Here is an example of such a function:

```
int sum(int x, int y) {
      return (x+y);
}
```

Now to initialize `fn` to the address of the sum, we can do the following:

- Make `fn` points to the address of sum

```
fn = &sum;
```

- Simply ignore the `&` as function names are just like array names, namely: they are pointers to the structure they are referring to.

```
fn = sum;
```

In the end, we can use the sum function in two ways.

```
int x = sum(10,12); /* direct call to the function */
int x = (*fn)(12,10); /* call to the function through a pointer */
```

Using Typedef's

The syntax of function pointers can sometimes be confusing. So we can use a typedef statement to make things simpler:

```
typedef (* fpointer)(argument list);
```

so we can define a variable of type `fpointer` as follows:

```
fpointer fp;
```

Now imagine you have a function that returns another function based on some conditions:

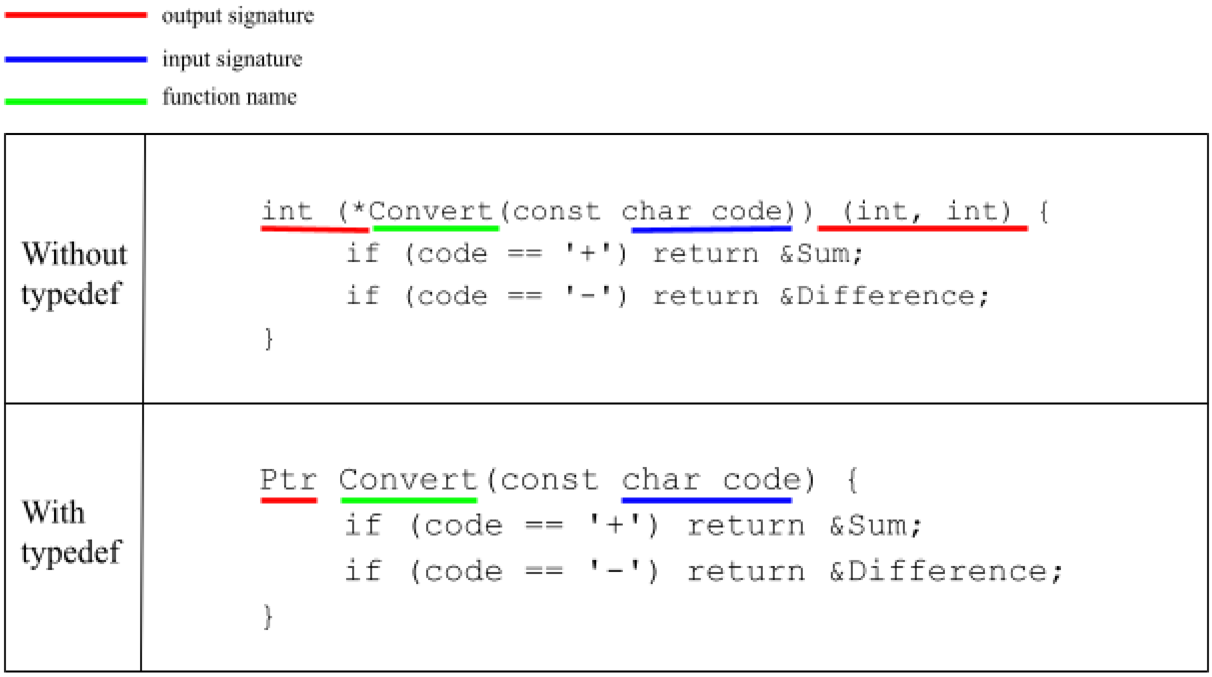
```
int (*Convert(const char code)) (int, int) {
     if (code = = '+') return &Sum;
     if (code = = '-') return &Difference;
}
```

The above function takes a `char` as an argument and returns a specific function pointer based on whether the char is + or – .

Both `Sum` and `Difference` are functions that essentially receive two numbers and return a single number, in this context we are working with ints, so we can do something like:

```
typedef int (*Ptr)(int,int);
```

And then simplify the Convert definition as follows:



🟥 output signature

🟦 input signature

🟩 function name

Without typedef	`int (*Convert(const char code)) (int, int) {`
	`        if (code == '+') return ∑`
	`        if (code == '-') return &Difference;`
	`}`

With typedef	`Ptr Convert(const char code) {`
	`        if (code == '+') return ∑`
	`        if (code == '-') return &Difference;`
	`}`

Linear search in C

Recap on linear search

In computer science, linear search or sequential search is a method for finding an element within a list. It sequentially checks each element of the list until a match is found or the whole list has been searched.

A linear search runs in linear time in the worst case, and makes at most n comparisons, where n is the length of the list

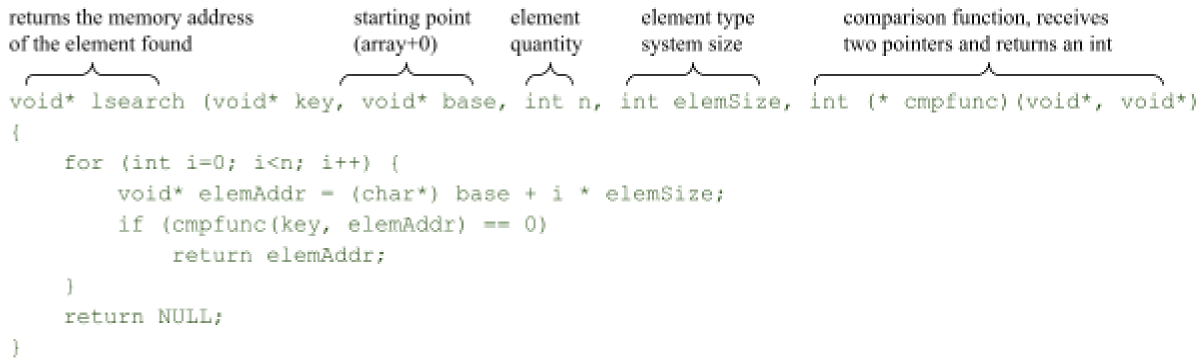
Basic algorithm

Given a list *L* of *n* elements with values or records L_0 .... L_{n-1}, and target value *T*, the following subroutine uses linear search to find the index of the target *T* in *L*.

1. Set *i* to 0.
2. If $L_i = T$, the search terminates successfully; return *i*.
3. Increase *i* by 1.
4. If $i < n$, go to step 2. Otherwise, the search terminates unsuccessfully.

More info at: https://en.wikipedia.org/wiki/Linear_search

Implementing a generic linear search


returns the memory address of the element found
starting point (array+0)
element quantity
element type system size
comparison function, receives two pointers and returns an int

```
void* lsearch (void* key, void* base, int n, int elemSize, int (* cmpfunc)(void*, void*)
{
    for (int i=0; i<n; i++) {
        void* elemAddr = (char*) base + i * elemSize;
        if (cmpfunc(key, elemAddr) == 0)
            return elemAddr;
    }
    return NULL;
}
```

Applying it over known `int` space requires us to write our comparing function for integers:

```
int intcmp (void* elem1, void* elem2)
{
    int* ip1 = (int*) elem1;
    int* ip2 = (int*) elem2;
    return *ip1 - *ip2;
}
```

Now putting everything together:

int_lsearch.c

```
#include <stdio.h>

void* lsearch (void* key, void* base, int n, int elemSize, int (*
cmpfunc)(void*, void*))
{
    for (int i=0; i<n; i++) {
        void* elemAddr = (char*) base + i * elemSize;
        if (cmpfunc(key, elemAddr) == 0)
            return elemAddr;
    }
    return NULL;
}

int intcmp (void* elem1, void* elem2)
{
    int* ip1 = (int*) elem1;
    int* ip2 = (int*) elem2;
    return *ip1 - *ip2;
}

int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    int key = 3;
    int size = sizeof(arr) / sizeof(arr[0]);
    int elemSize = sizeof(arr[0]);
    int* found = (int*) lsearch(&key, arr, size, elemSize,
intcmp);
    if (found != NULL)
        printf("Found: %d\n", *found); //remember to never
dereference a NULL pointer
    else
        printf("Not found\n");
    return 0;
}
```

> Found: 3

In case we want to search for a specific string in an array of strings (array of pointers to pointers), we can borrow the `Strcmp` function we implemented in part 1 and do something like:

```
char* note = "Eb";
char* notes[] = { "Ab", "F#", "B", "Gb", "D" };
char** found = lsearch(&note, notes, 5, sizeof(char**), Strcmp);
```

Approaching C++

The idea of a class

First, let us establish that **`Structs`** in C and **`Classes`** in C++ are identical in all ways except for the default access modifier: for a `struct` the default is **public**, whereas for a class it is **private**. There's no other difference as far as the language is concerned.

The arrow operator (->) vs the dot (.) operator

dot (.)

The dot operator is used to access the content of a member of a struct (or class in C++) directly when you have an instance of the struct or class.

Example: struct_instance.member

arrow (->)

The arrow operator is used to access the content of a member of a struct (or class) when you have a pointer to the struct (or class).

The arrow operator is syntactic sugar for dereferencing the pointer before accessing the member content via the dot operator.

Example: ptr->member is equivalent to (*ptr).member

Implementing a stack data structure for ints

Before we begin, we need to know that in C it is a common practice to aggressively separate implementation from behavior. This means that we should first create a `.h` file to store the implementation (interface) of the stack structure and then a `.c` file with the actual behavior.

`stack.h`

```
typedef struct {
    int *elems;
    int logicLen;
    int allocLen;
} Stack;

void StackNew(Stack* s);
void StackDispose(Stack* s);
void StackPush(Stack* s, int value);
int StackPop(Stack* s);
```

`stack.c`

```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include "stack.h" //assuming stack.h is in the
same directory of stack.c


void StackNew(Stack* s) {
    s->logicLen = 0;
    s->allocLen = 4;
    s->elems = malloc(4 * sizeof(int));
    assert(s->elems != NULL);
}

void StackDispose(Stack* s) {
    free(s->elems); //returns memory to the heap
}

void StackPush(Stack* s, int value) {
    if(s->logicLen == s->allocLen) {
        s->allocLen *= 2;
        s->elems = realloc(s->elems,
s->allocLen*sizeof(int));
        assert(s->elems != NULL);
    }
    s->elems[s->logicLen] = value;
    s->logicLen++;
}

int StackPop(Stack* s) {
    assert(s->logicLen > 0);
    s->logicLen--;
    return s->elems[s->logicLen];
}

int main() {
    Stack s;
    StackNew(&s);
    StackPush(&s, 1);
    StackPush(&s, 2);
    StackPush(&s, 3);
    assert(StackPop(&s) == 3);
    assert(StackPop(&s) == 2);
    assert(StackPop(&s) == 1);
```

```
                    StackDispose(&s);
                    printf("All Good\n");
                    return 0;
                }
```

Note that `assert()` is just a macro we use here to ensure we get the *memblock* pointer in the end.

Quick words about realloc()

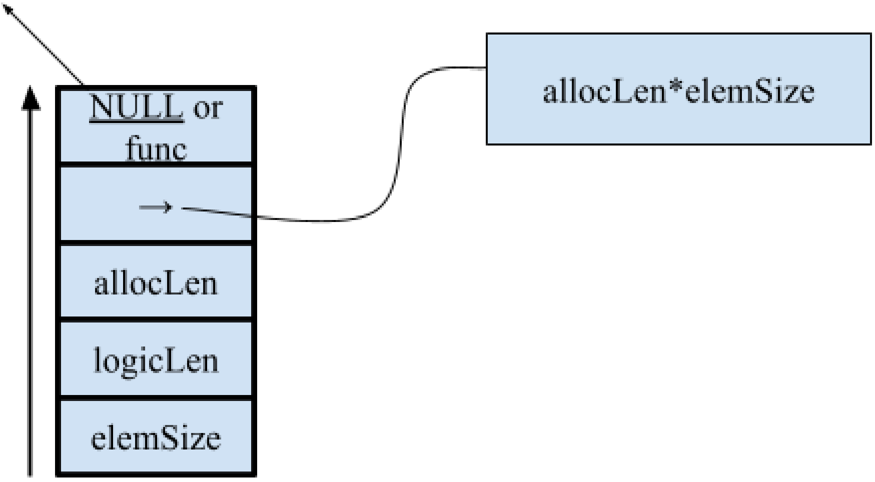
- Deallocate old array and returns a resized array;
- Has the advantage of checking if there is available space to contiguously extend the memory in the heap for `s->elems`;
- There is no similar function in c++;
- Fun fact: `malloc(size) ≡ realloc(Null, size)`

Implementing a generic stack data structure

`stack.h`

```
typedef struct {
    void* elems;
    int elemSize;
    int logicLen;
    int allocLen;
    void (*freefunc)(void*);
} Stack;

void StackNew(Stack* s, int elemSize, void (*freefunc)(void*));
void StackDispose(Stack* s);
void StackPush(Stack* s, void* elemAddr);
void StackPop(Stack* s, void* elemAddr);
```

If the stack is implemented for one of the basic types there is no need for a custom free function



memory allocation for Stack struct

`stack.c`

```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "stack.h" //assuming stack.h is in the same directory of stack.c


void StackNew(Stack* s, int elemSize, void(*freefunc)(void*)) {
    if(elemSize <= 0) {
        assert(0);
    }
    s->elemSize = elemSize;
    s->logicLen = 0;
    s->allocLen = 4;
    s->elems = malloc(4*elemSize);
    s->freefunc = freefunc;
    assert(s->elems != NULL);
}

/* `static` is analogous to `private` in c++; in C it marks the file as internal linkage */
static void StackGrow(Stack* s) {
    s->allocLen *= 2;
    s->elems = realloc(s->elems, s->allocLen * s->elemSize);
}
```

```
void StackPush(Stack* s, void* elemAddr) {
    if(s->logicLen == s->allocLen)
        StackGrow(s);
    void* target = (char*)s->elems + (s->logicLen * s->elemSize);
    memcpy(target, elemAddr, s->elemSize);
    s->logicLen++; //sets next available index
}

/* note that elemAddr gets modified externally via StackPop
internal operations */
void StackPop(Stack* s, void* elemAddr) {
    s->logicLen--;
    void* source = (char*)s->elems + (s->logicLen * s->elemSize);
    memcpy(elemAddr, source, s->elemSize);
}

/* we have to make sure to clear all structures whenever
applicable */
void StackDispose(Stack* s) {
    if(s->freefunc != NULL) {
        for(int i=0; i<s->logicLen; i++) {
            s->freefunc((char*)s->elems + (i * s->elemSize));
        }

    }
    free(s->elems);
}

void StringFree(void* elem) {
    free(*(char**)elem);
}

int main(){
    const char* friends[] = {"Al", "Bob", "Carl"};
    Stack stringStack;
    StackNew(&stringStack, sizeof(char**), StringFree);
    for(int i=0; i<3; i++) {
        char* copy = strdup(friends[i]);
        StackPush(&stringStack, &copy); //push the pointer to the
string copy (char**) onto the stack
    }
    char* name;
    printf("Popping elements:\n");
    for(int i=0; i<3; i++) {
        StackPop(&stringStack, &name);
        printf("%s\n", name);
        free(name);
    }
    StackDispose(&stringStack); //agnostic of stack content by time of execution
}
```

Pause a bit to ponder on the above implementation, make sure you understand and agree with what each line is doing, and don't hesitate to go back to part 1 if you need to review something.

Quick words on memory disposal

Disposing basic types

If you dynamically allocate memory for basic types (*e.g int, float, char, etc)*, `free()` will correctly deallocate the memory without any additional steps needed:

```
int *p = malloc(sizeof(int));
// Use p...
free(p);  // No additional action needed
```

Disposing structures

For simple structures, like a fraction struct or any struct that only contains basic types, `free()` will deallocate the memory for the structure itself, but if the structure contains pointers to dynamically allocated memory (*e.g char**, pointers to structs, structs with pointers, etc*), you'll need to manually `free()` those inner allocations before freeing the structure:

```
typedef struct {
    char* name;
    int age;
} Person;

Person *p = malloc(sizeof(Person));
p->name = malloc(100 * sizeof(char));  // Dynamically allocate memory for name

// Use p...

free(p->name);  // Free the inner allocation first
free(p);        // Then free the structure itself
```

In this case, failing to free `p->name` before freeing `p` would lead to a memory leak. C doesn't automatically traverse the structure to free any internal pointers, so you have to handle that manually.

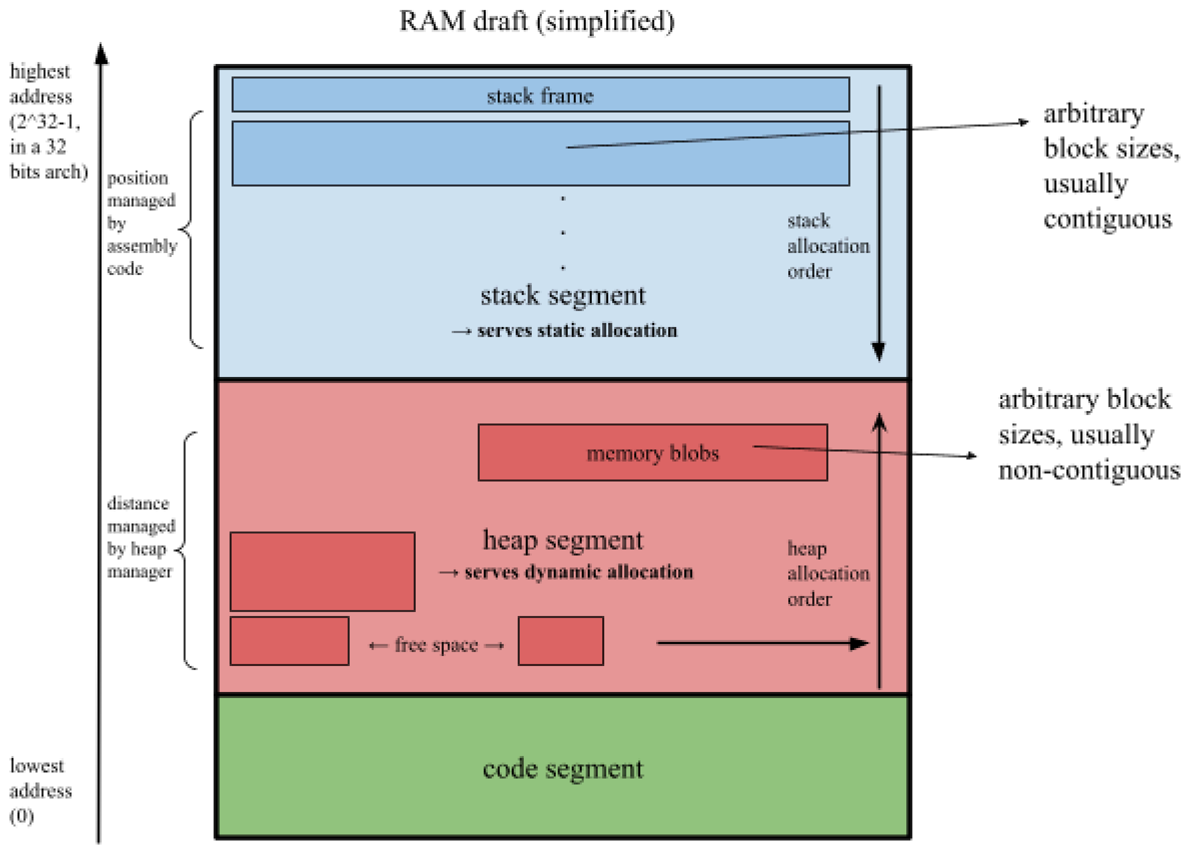
That is why we had to create a custom `StringFree()` function for our `stringStack` use case

.

The RAM memory

RAM is a common computing acronym that stands for random-access memory. It is often referred to as PC memory or simply memory. In essence, RAM is your computer's short-term memory, where data needed by the CPU to run applications and open files is stored for quick access.

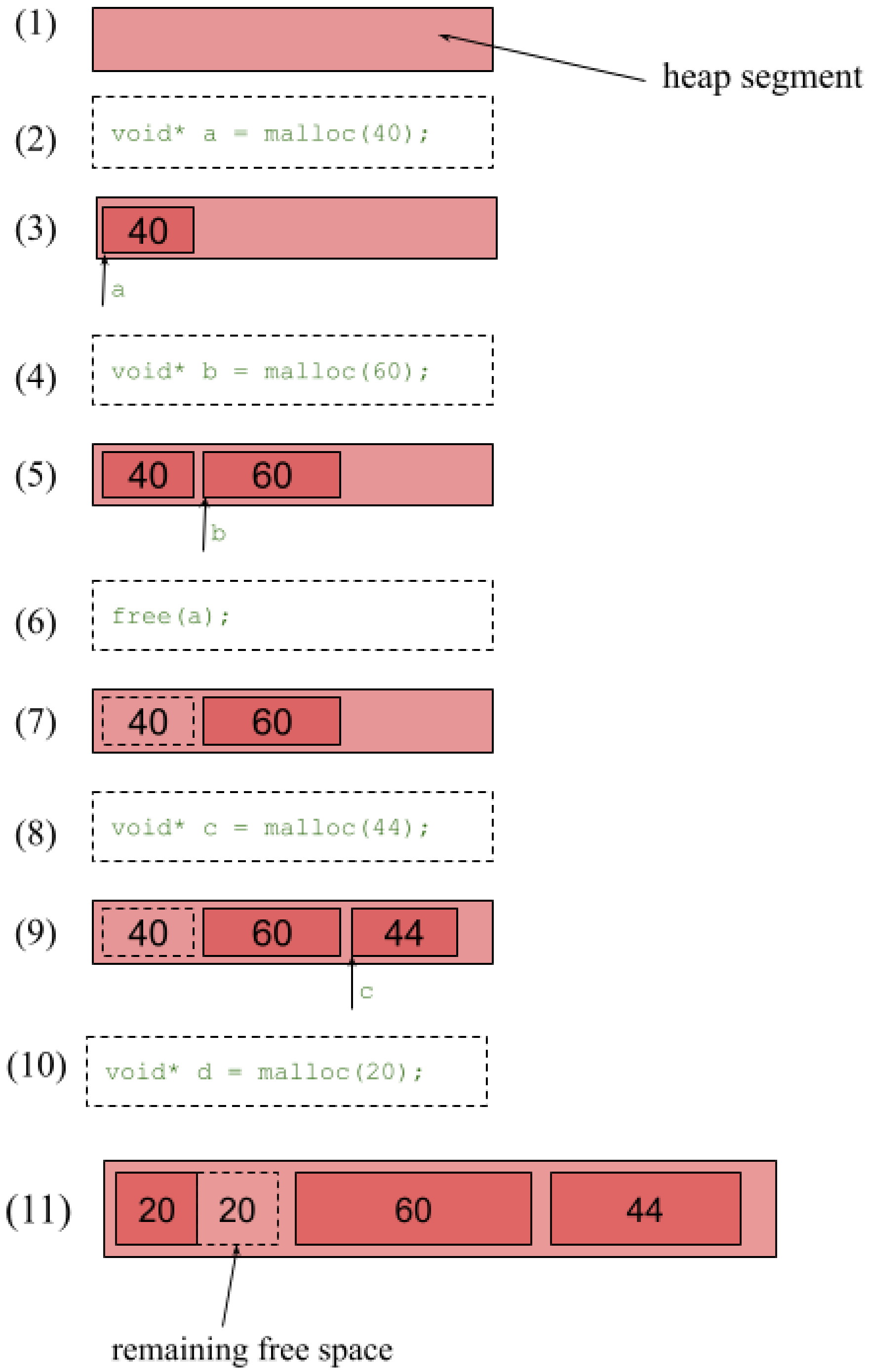
When memory is allocated—either statically on the stack via stack frames or dynamically on the heap—we are utilizing RAM. It is also referred to as primary storage.


RAM draft (simplified)
highest address (2^32-1, in a 32 bits arch)
position managed by assembly code
stack frame
stack allocation order
stack segment
→ serves static allocation
arbitrary block sizes, usually contiguous
memory blobs
arbitrary block sizes, usually non-contiguous
distance managed by heap manager
heap segment
→ serves dynamic allocation
heap allocation order
← free space →
code segment
lowest address (0)

Despite the lack of formal constraints, stack frames allocation are usually contiguous, and even if not contiguous in physical memory they are usually contiguous in virtual memory, more info at https://stackoverflow.com/questions/5086577/is-stack-memory-contiguous

Unlike stack frames, heap allocations are typically not contiguous. Heap memory is often fragmented and can consist of blocks that are scattered throughout both physical and virtual memory, depending on the allocation and deallocation patterns.

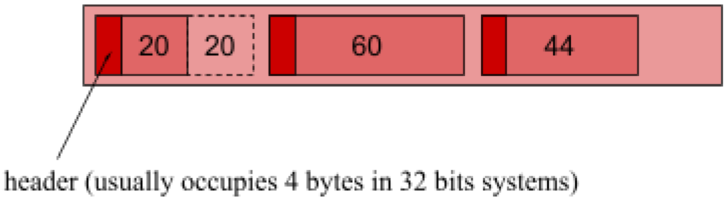
Zeroing in the heap segment with malloc()



```
void* a = malloc(40);
void* b = malloc(60);
free(a);
void* c = malloc(44);
void* d = malloc(20);
```

Zeroing a bit more

When `` `int *arr = malloc(40*sizeof(int))` `` is executed, the heap manager actually accommodates more than 160 bytes of memory. Along with the reserved memory blob lies a header that stores additional information, like memory footprint, etc.



General header information

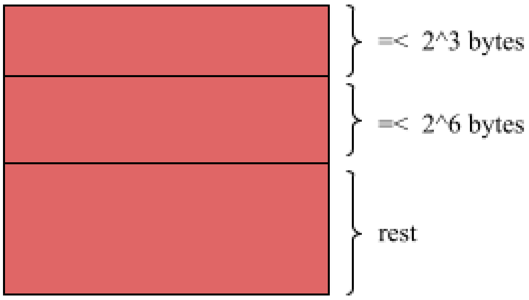
- **Block size**: The header usually stores the size of the allocated block;
- **Status**: It may include a flag indicating whether the block is free or in use;
- **Pointers to neighboring blocks**: In some memory allocators (especially those implementing a free list), the header might include pointers to the next and previous blocks in the list.

When `malloc()` runs, the memory management system stores the leading address of that memory blob and the address to its header in a symbol table which `free()` eventually relies on in order to know how much memory to deallocate. Hence, one cannot simply give any pointer to `free()`, it needs to be the same pointer that was handed back from `malloc()` in the first place.

e.g: `free(arr+1)` will not work as only the leading address of the array `(arr+0)` is kept at the symbol table by `malloc()` with a header reference that holds how much memory was allocated to the whole blob. The same is true for string arrays.

Two examples of heap segmentation are:

1. divided by groups of block sizes



2. big heap (basically the one shown in the RAM draft)

Important reminder: since there are many different heuristics on memory management and heap segmentation per OS, memory-specific interactions shouldn't be relied upon by the client. The way the heap is traversed when searching for available space is also arbitrary to implementation trade-offs

memcopy() vs memove() and the overlapping issue

Prototype: `void *memcpy(void *dest, const void *src, size_t n);`

- dest: Pointer to the destination memory location;
- src: Pointer to the source memory location;
- n: Number of bytes to copy from the source to the destination.

`memcpy()` assumes that the source (`src`) and destination (`dest`) memory areas do not overlap. The function directly copies n bytes from `src` to `dest` without performing any checks to see if these two memory regions might overlap.

If `src` and `dest` overlap, using `memcpy()` can lead to undefined behavior. This means that the outcome of the copy operation is unpredictable—data corruption or other unexpected issues might occur because the source data might be overwritten before it is fully copied.

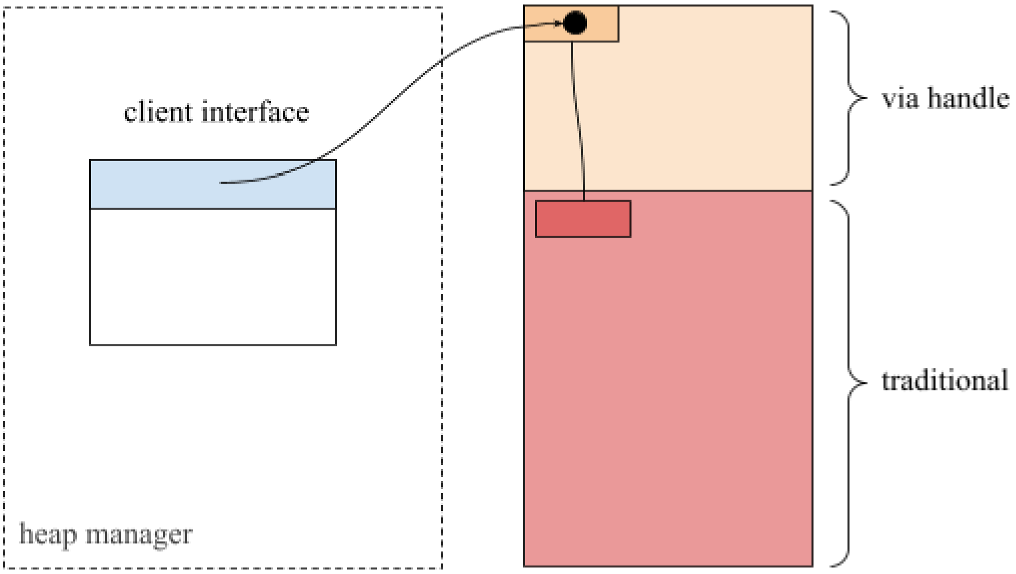
If there is a possibility that the memory regions might overlap, the `memmove()` function should be used instead. Unlike `memcpy()`, `memmove()` is designed to handle overlapping memory areas safely and does so with a slightly higher performance cost than *memcpy()*.

Handles and heap compaction

Indirect approaches, like using "handles" or "indirect pointers," can allow for heap compaction, which is a process where fragmented blocks of memory are consolidated to make larger contiguous blocks available.

Handles

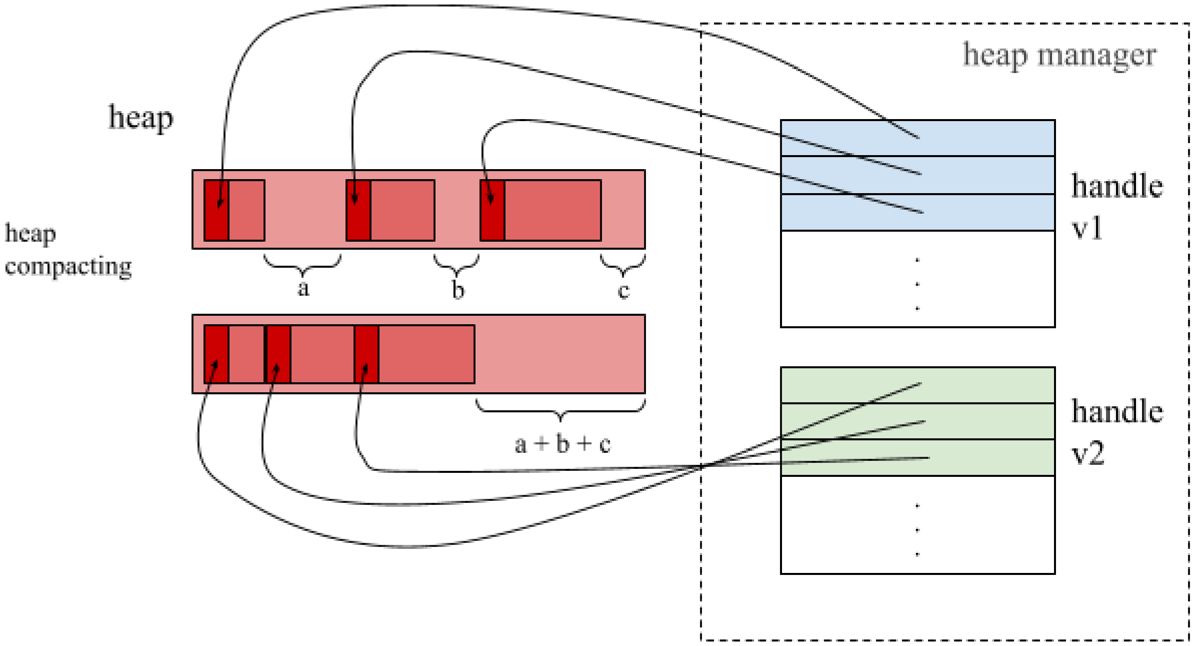
A handle is essentially a level of indirection where a pointer (or reference) does not directly point to the memory location of the data but instead points to a handle (a fixed location). This handle, in turn, points to the actual data. Therefore, the pointers handed to the client are two hops away from the actual data.



In other words, the handle serves as a lookup table for client reference that points to the header of a memory segment and can be updated during heap optimization processes.

Heap Compaction

When heap memory becomes fragmented due to allocations and deallocations, the memory manager can move the actual data to a new location in order to create larger contiguous blocks of free memory. Since the application code only references the handle and not the direct memory address, the memory manager can update the handle to point to the new location of the data, without requiring the application code to change.



The STACK memory

The stack pointer

The stack pointer is a special CPU register that holds the memory address of the top of the stack. it essentially moves step-by-step along with every piece of data that is allocated on the stack during the function call and execution process.

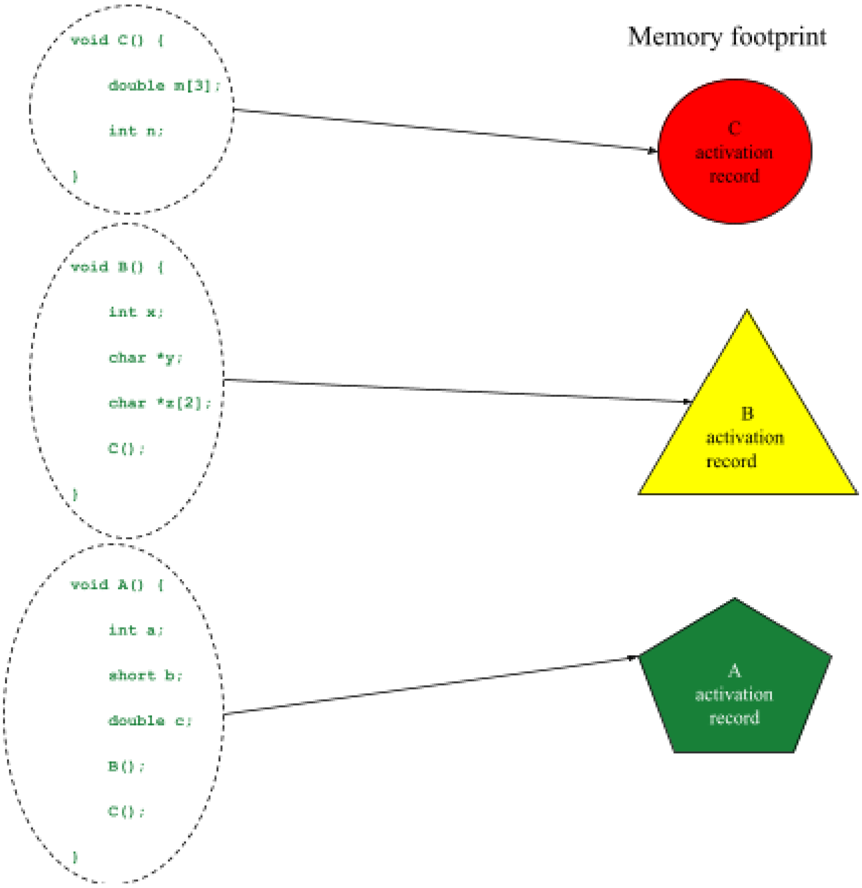
Naming conventions

- **SP (Stack Pointer)**: Used in 16-bit architectures.
- **ESP (Extended Stack Pointer)**: Used in 32-bit architectures.
- **RSP (Register Stack Pointer)**: Used in 64-bit architectures.

Activation record

An activation record is a logical concept used to describe the data structure that holds all the information necessary to execute a function. It's the formal name for the information held in a stack frame. Mind that the exact contents and layout of the stack vary by processor architecture and function call convention (more info at: https://manybutfinite.com/post/journey-to-the-stack/)

Suppose we have:



```
void C() {
    double m[3];
    int n;
}
```

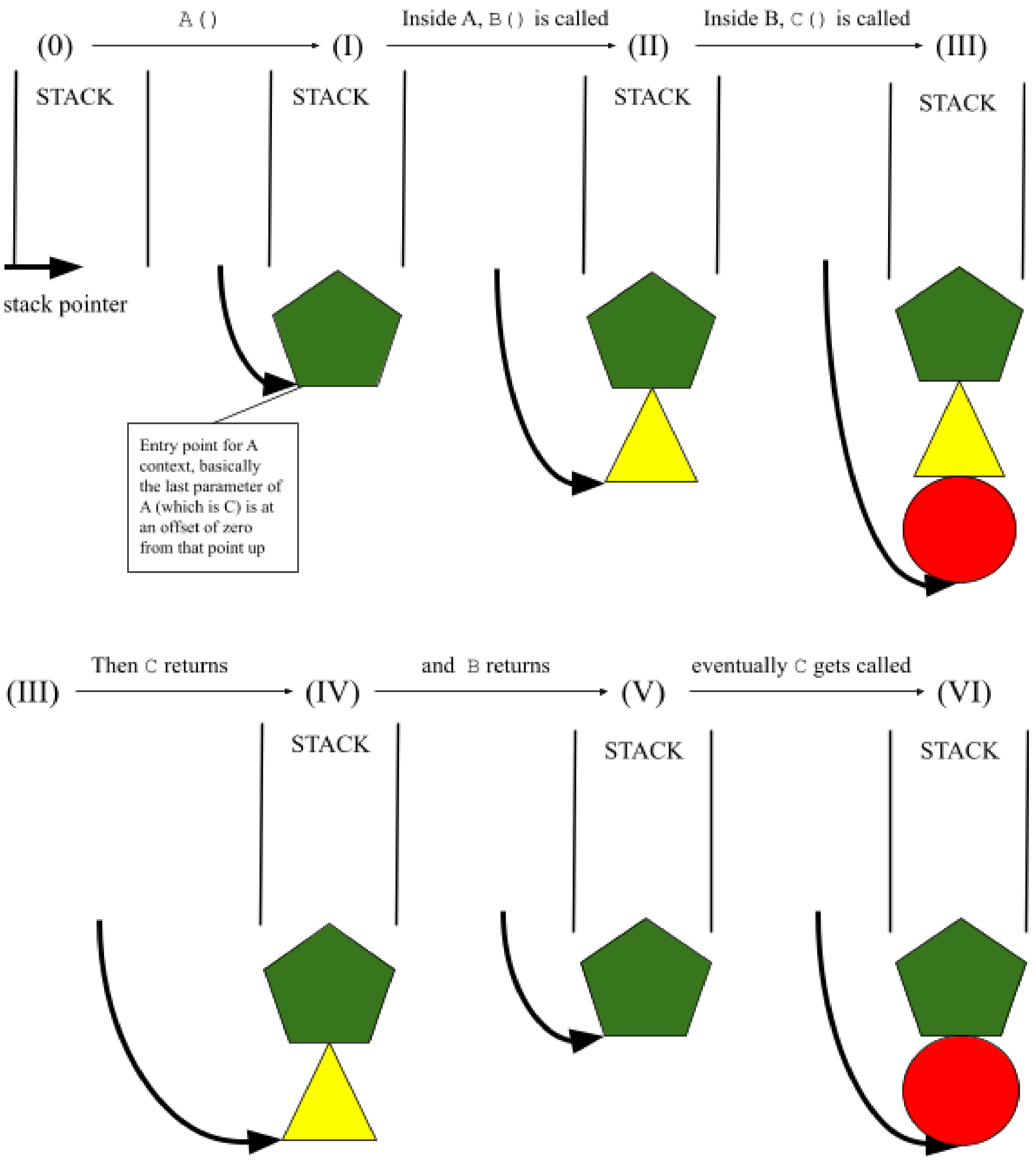
```
void B() {
    int x;
    char *y;
    char *z[2];
    C();
}
```

```
void A() {
    int a;
    short b;
    double c;
    B();
    C();
}
```

When:

```
int main() {

    A();

    return 0;

}
```

is executed, the stack pointer descends (higher memory address → lower memory address) by the amount given by A's activation record memory footprint, then B's and C's, without losing track of the higher frames.



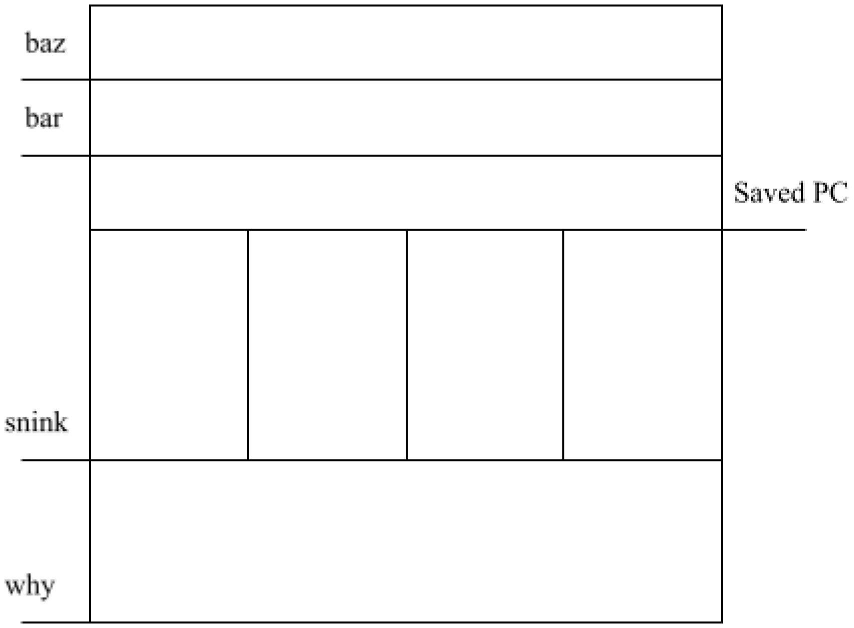
Entry point for A context, basically the last parameter of A (which is C) is at an offset of zero from that point up

Zeroing on the activation record

Now suppose we have:

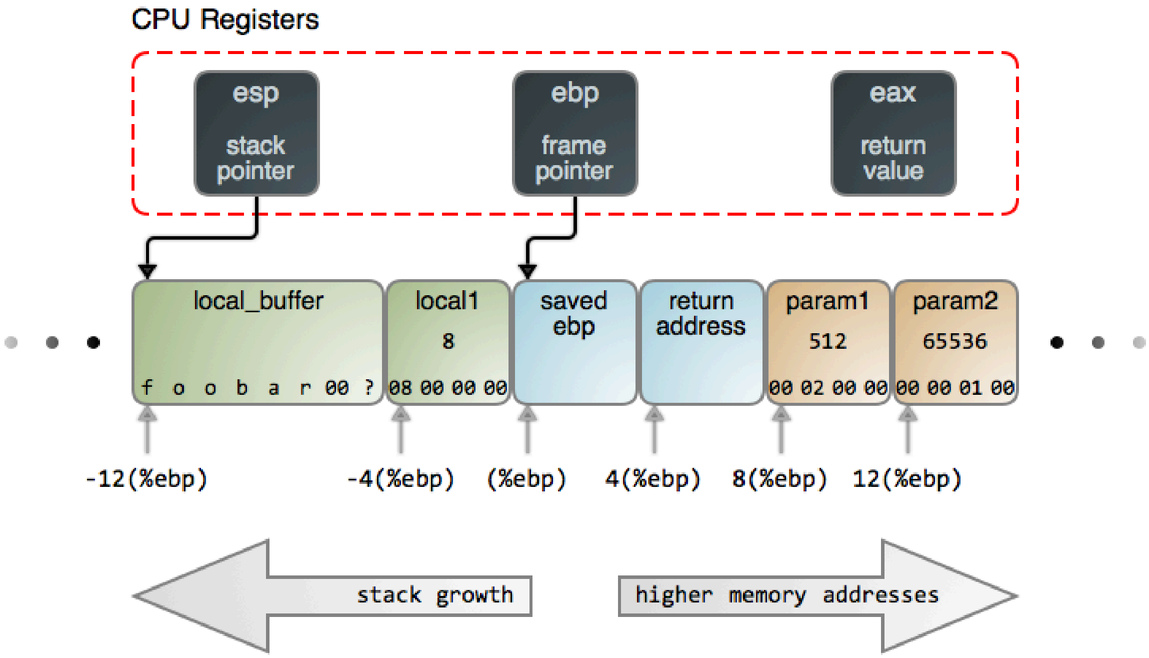
```
void foo(int bar, int *baz) {
    char snink[4];
    short *why;
}
```

Foo's activation record of would be filled top-down starting right-to-left on the function signature:



Where ***Saved PC*** is also known as ***Program Counter*** or ***Return Address*** and it stores the address of the instruction to be executed upon the function's exit.

It's worth mentioning that some architectures like Intel's x86 also contain something called the ***Frame Pointer***, ***Base Pointer*** or ***EBP***, which sits on top of the *Saved PC* pointing to a fixed location within the stack frame of the function currently running providing a stable reference point (base) for access to arguments and local variables, traditionally for such architectures both the base pointer (*EBP*) and the return address (*Saved PC*) are saved on the stack during a function call.



Intel x86 stack example, source: https://manybutfinite.com/post/journey-to-the-stack/

In modern compilers and architectures, there's a technique called ***frame pointer omission*** (***FPO***). This technique eliminates the use of the *base pointer (EBP)* to save registers, making the function more efficient by freeing up ***EBP*** for general-purpose use. In these cases:

- Only the *Saved PC* (return address) is pushed onto the stack.
- The stack is managed solely by the *stack pointer (ESP*), and offsets from *ESP* are used to access local variables and parameters.

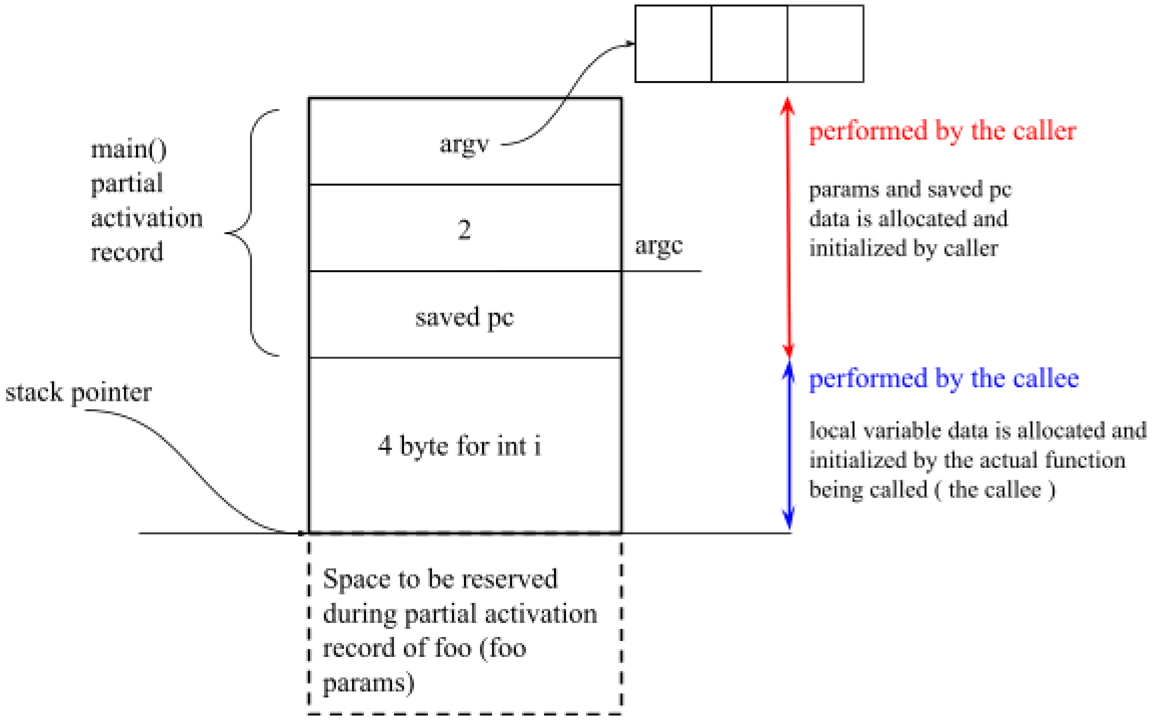
This results in slightly faster and smaller code because there's no need to preserve or restore the base pointer, and fewer instructions are used during function calls and returns.

Note that when I say "pushed", I mean added to the top of the stack. Since the stack grows downward in memory addresses, pushing an item essentially means placing it at a lower memory address than the previous "top" item.

Continuing, suppose we now have:

```
int main(int argc, char** argv) {

    int i = 4;
    foo(i, &i);
    return 0;

}
```

In such case, one activation record diagram for it could be:


main()
partial
activation
record
argv
2
argc
saved pc
stack pointer
4 byte for int i
Space to be reserved during partial activation record of foo (foo params)
performed by the caller
params and saved pc data is allocated and initialized by caller
performed by the callee
local variable data is allocated and initialized by the actual function being called (the callee)

More information about *argc* and *argv* can be found here: https://stackoverflow.com/questions/3024197/what-does-int-argc-char-argv-mean

Partial activation record of main():

During the partial activation record for main (performed by main's caller), we have parameters and program counter initialization followed by control delegation, namely:

1. caller makes room for main stack data
2. `argc` and `argv` are initialized (pushed into main stack) by the caller;
3. call <main> instruction is executed by the runtime environment, pushing the saved PC (return address) onto the stack and then transfers control to main()

Once main's partial activation is complete, main's function prologue allocates space for its local variables and initializes them.

Partial activation record of foo():

During the partial activation record for foo (performed by main), the cycle repeats itself and we also have parameters and program counter initialization followed control delegation, namely:

1. main makes room for foo stack data
2. The parameters passed to `foo()` within `main` (i and &i) are pushed by the caller (main) onto foo's stack;
3. Main executes call <foo> instruction, which pushes the saved PC (return address) onto the stack and then transfers control to foo()

Once foo's partial activation is complete, foo's function prologue allocates space for its local variables and initializes them.

Taking M as a notation for "all RAM", we could rewrite the partial activation record of foo() with a mock assembly language to help grasp the flow:

```
...

SP = SP - 4;          # Allocate space for main's local variable
'i'
M[SP] = 4;            # Initialize 'i' with the value 4

# -------------------------
# Partial Activation of foo
# -------------------------

R1 = SP + 8;          # Load the address (&i) of `i` into register
R1
                      # (first element of argv)
R2 = M[SP+8];         # Load the value of `i` into register R2
SP = SP - 8;          # Allocate space for foo's arguments
```

```
# -------------------------
# Marks the bottom of the full activation record for main() stack
# and beginning of the activation record for the foo() stack
# -------------------------

M[SP] = R2;          # foo's first argument left to right (in the
                     # "top" of the pile, since the stack grows
                     # downwards)

M[SP+4] = R1;        # foo's second argument left to right


# Control transfer to foo()
# automatically pushes the return address (Saved PC) onto the
stack

call <foo>
```

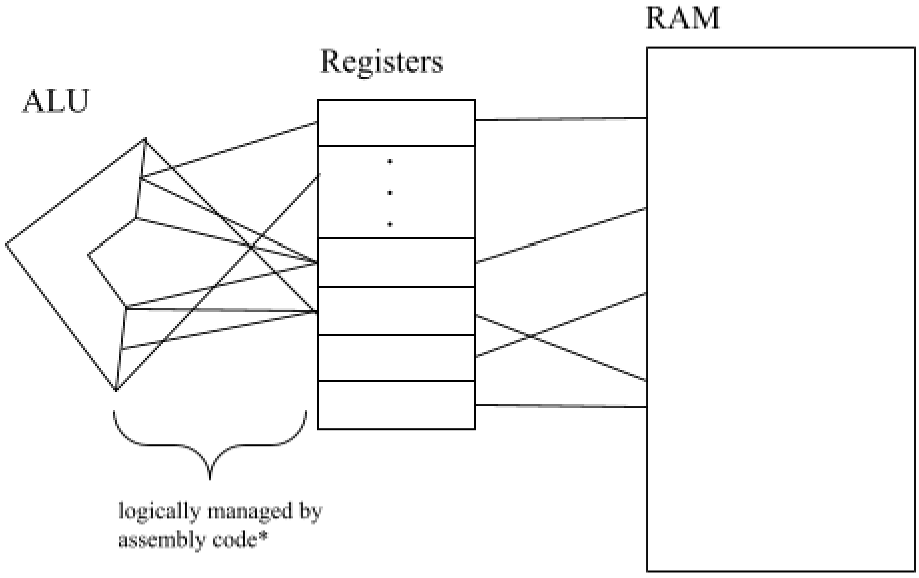
Assembly

Assembly code (or assembly language) is a low-level programming language that is closely related to machine code, the instructions directly executed by a computer's CPU. However, instead of writing in binary or hexadecimal (which machine code uses), assembly code allows programmers to use symbolic names and mnemonics that are easier to understand. Each assembly instruction corresponds to a single machine instruction, providing control over hardware at a detailed level.

Registers

Registers are small and fast storage locations within the CPU that hold data temporarily during instruction execution. The relationship between assembly code and registers is tight because assembly instructions often involve manipulating data stored directly in these registers.

The CPU constantly moves data between RAM (which is slow compared to registers) and registers**.** Data and instructions are loaded from RAM into registers, processed by the CPU, and then results may be written back to RAM.



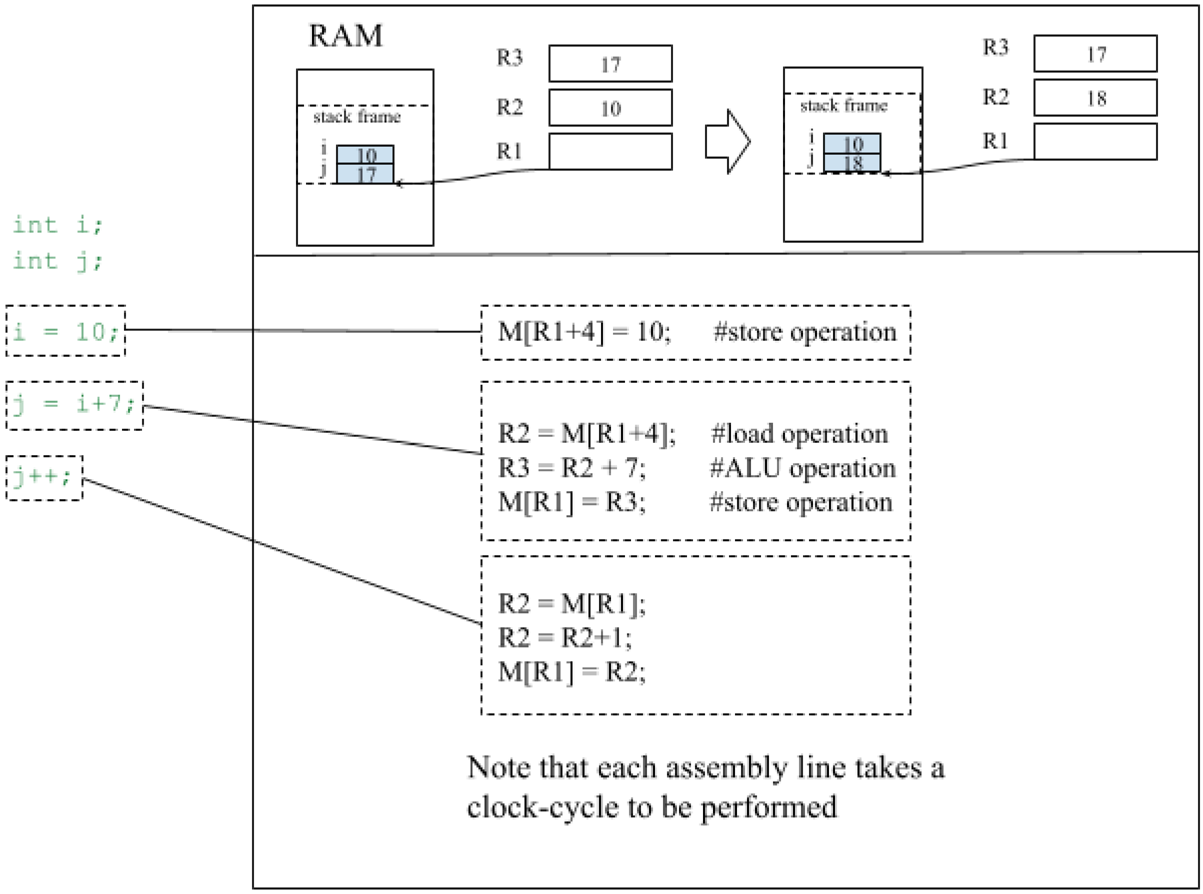
*ALU is electrically in touch with registers, but not with RAM

C to assembly

Continuing with our mock assembly language where M is a notation for "all RAM", suppose we have the following C code:

```
int i;
int j;

i = 10;
j = i+7;
j++;
```

How would it translate to our mock assembly? Somewhat like that:



```
int i;
int j;
i = 10;
j = i+7;
j++;
```

```
M[R1+4] = 10;     #store operation
```

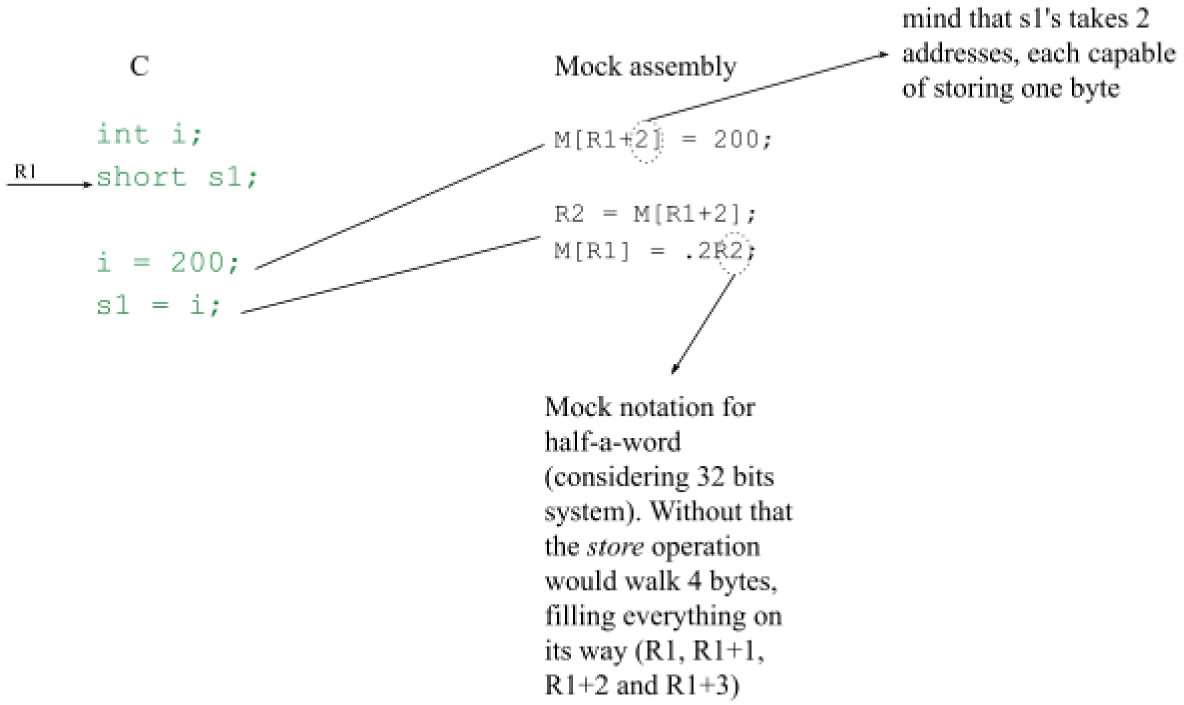
```
R2 = M[R1+4];    #load operation
R3 = R2 + 7;     #ALU operation
M[R1] = R3;      #store operation
```

```
R2 = M[R1];
R2 = R2+1;
M[R1] = R2;
```

Note that each assembly line takes a clock-cycle to be performed

Half word notation

Registers are typically designed to hold a single *word* and its operations are made in word-wide steps.



C

```
int i;
short s1;

i = 200;
s1 = i;
```

Mock assembly

```
M[R1+2] = 200;

R2 = M[R1+2];
M[R1] = .2R2;
```

mind that s1's takes 2 addresses, each capable of storing one byte

Mock notation for half-a-word (considering 32 bits system). Without that the *store* operation would walk 4 bytes, filling everything on its way (R1, R1+1, R1+2 and R1+3)

To properly store s1 in RAM at `s1 = i` we have to cast/truncate the 200 int (4 bytes) to short (2 bytes) by using a different mnemonic that operates only half-a-word.

Dereferencing

Quick review on stack memory allocation
In most architectures (like x86), the stack grows from higher memory addresses to lower ones. This means that when you allocate space on the stack, the stack pointer (SP) decreases.

For example:

```
SP = SP - 4   ; # Allocate 4 bytes on the stack
MP[SP] = 5    ; # Store the integer value 5 starting at address SP
```

After `SP = SP - 4`, the stack pointer now points to the start of the allocated 4-byte space. Writing a value to this space typically involves storing data starting from the current `SP` address up to `SP + 3` (usually with least significant byte first, aka little-endian).

Now suppose we have:

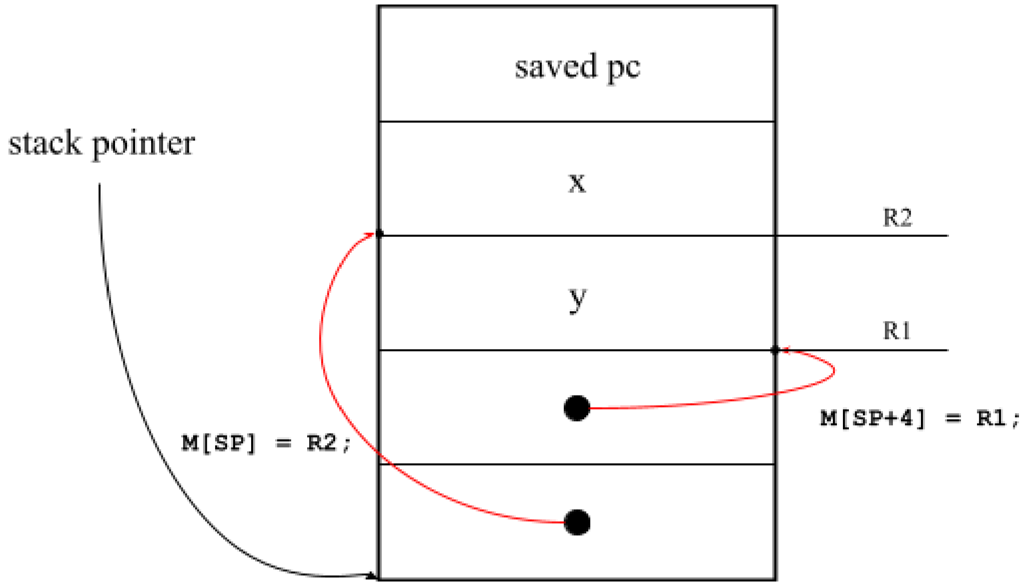
```
void foo() {
    int x;

    int y;

    x = 11;

    y = 17;

    swap(&x, &y);

}
```

When called this would translate to

```

...

SP = SP - 8;         # Allocate space for foo's locals
M[SP+4] = 11;        # Initialize locals
M[SP] = 17;

# -------------------------
# Partial Activation of swap
# -------------------------

R1 = SP;             # &y
R2 = SP + 4;         # &x
SP = SP - 8;         # allocate memory for swap args

# -------------------------
# Marks the bottom of the full activation record for foo() stack
# and beginning of the activation record for the swap() stack
# -------------------------

M[SP] = R2;          # initialize swap args
M[SP+4] = R1;


# Control transfer to swap()
# automatically pushes the return address (Saved PC) onto the
stack

call <swap>

SP = SP + 8;             # deallocation for swap
SP = SP + 8;             # deallocation for foo
```

```
RET;
...
```

Take a moment to grasp how swap args are initialized with x and y addresses:



Anatomy of a dereference

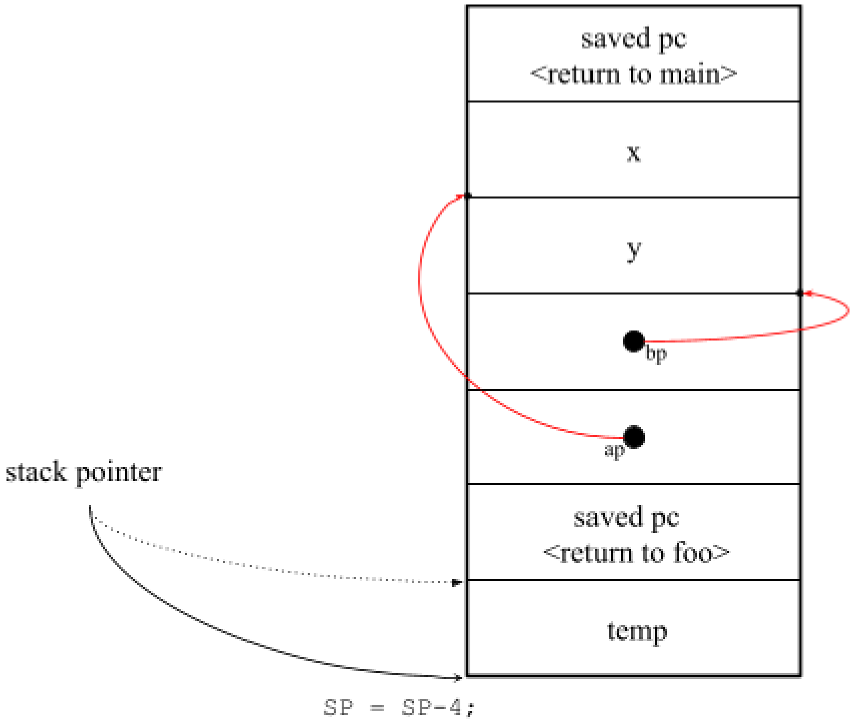
Suppose swap is implemented as follow:

```
void swap (int *ap, int *bp) {
    int temp = *ap;
    *ap = *bp;
    *bp = temp;
}
```

Now let’s examine carefully how dereferencing would translate into assembly code:

```
...

SP = SP - 4;          # Allocate space for swap local variable
`temp`

...
```

Extending our previous diagram, it becomes somewhat like that:



Continuing with the assembly

```
...

# --------------------------
# Dereferencing ap
# --------------------------
R1 = M[SP+8];          # loads address of ap
R2 = M[R1];            # loads value of ap (*ap)
```

~~~
M[SP] = R2;           # stores ap value into `temp`


# -------------------------
# Dereferencing bp
# -------------------------

R1 = M[SP+12];        # loads address of bp
R2 = M[R1];           # loads value of bp (*bp)

# -------------------------
# Swapping values
# -------------------------

R3 = M[SP+8];         # loads address of ap
M[R3] = R2;           # replace ap value with bp value (*ap = *bp)
R1 = M[SP];           # loads address of `temp` (now ap value)
R2 = M[SP+12];        # loads address of bp
M[R2] = R1;           # replace bp value with ap value (*bp = temp)

SP = SP + 4;          # returns SP to saved pc
RET;

...

```
~~~

C++ references

What is a reference?

In C++, a reference is an alias for an existing variable, which is different from a pointer.

- A pointer is a variable that stores the memory address of another variable;
- A reference can be thought of as a constant pointer, by using references you lose the ability of reassociation.

For further comparison see https://stackoverflow.com/questions/57483/what-are-the-differences-between-a-pointer-variable-and-a-reference-variable

Syntax and Declaration:

- Declared using the `&` symbol:

  ```
  int x = 10;
  int& ref = x;
  ```

Binding

- A reference must be initialized at the time of declaration and cannot be reseated to refer to another variable:

```
int a = 5, b = 10;
int& ref = a;      // ref is bound to a
ref = b;           // assigns the value of b to a, does NOT change the reference.
```

Dereferencing

- Automatically dereferences the variable it refers to. No explicit * is needed to access or modify the referenced variable (the compiler will apply the * Operator for you).
- Example:

```
int x = 10;
int& ref = x;
ref = 20;  // modifies x directly.
```

Never null

- A reference cannot be null, as it must always refer to a valid object.

Physical memory Address

- References don't inherently have their own address in the stack; their address is typically that of the variable they refer to. However, in specific cases like function arguments passed by reference or class member references, references might exist as entities on the stack. For more information see https://stackoverflow.com/questions/45821678/is-always-the-address-of-a-reference-equal-to-the-address-of-origin

Assembly operation

Suppose we have

```
int y = 17;
int &z = y;
int *z = &y;
```

```

Despite the difference of what you can do with a reference and a pointer within the language, both `int &z = y;` and `int *z = &y;` in assembly translate to the same operation.

---

## Executable generation

### Preprocessing

**What is a preprocessor?**

They are C or CPP modules that reads the code before compilation looking for `#define`s and `#include`s replacing `key:values` along, like almost pure token search and text replace , and in the end outputs the `{code}` without `#define`s and `#include`s (that got replaced with the segment they represent).

- For `#define`, it performs token-based substitution (replacing a key with its corresponding value);
- For `#include`, the preprocessor recursively replace the whole line containing with the file content it is targeting; file could be local headers accessed with " " like `#include "file.h"` or system's like `#include <stdio.h>`

Mind that in C (and C++), #include statements usually* bring declarations (function prototypes, struct definitions, etc.) into your compilation unit. The function implementations (i.e., the actual compiled machine code) live elsewhere and it isn't pulled into your program until link time.

*Inline functions can sometimes be placed directly in header files because the compiler can insert (inline) the code for that function into your source. But this is a special case, and it's typically used for small, performance-critical functions. In C++, templates also must be provided in headers so the compiler can generate specialized code for the types you use. But again, the actual compiled instructions for these instantiations still end up in object files, which are finally stitched together by the linker.

**Example**

Input Code:

```
    #define SQUARE(x) ((x) * (x))

    int main() {

```
        int result = SQUARE(5);

        return result;

    }
```

After Preprocessing:

```

    int main() {
        int result = ((5) * (5));
        return result;
    }

```

This preprocessed code is what the compiler actually works with. It's purely text-based replacement and doesn't perform semantic checks during this stage (e.g., it won't validate the correctness of #define values).

The example above is also known as a **macro**, notice that it also gets text-search-replaced by the preprocessor but with a text that will eventually get evaluated as an expression (i.e `5*5`) in compile-time.

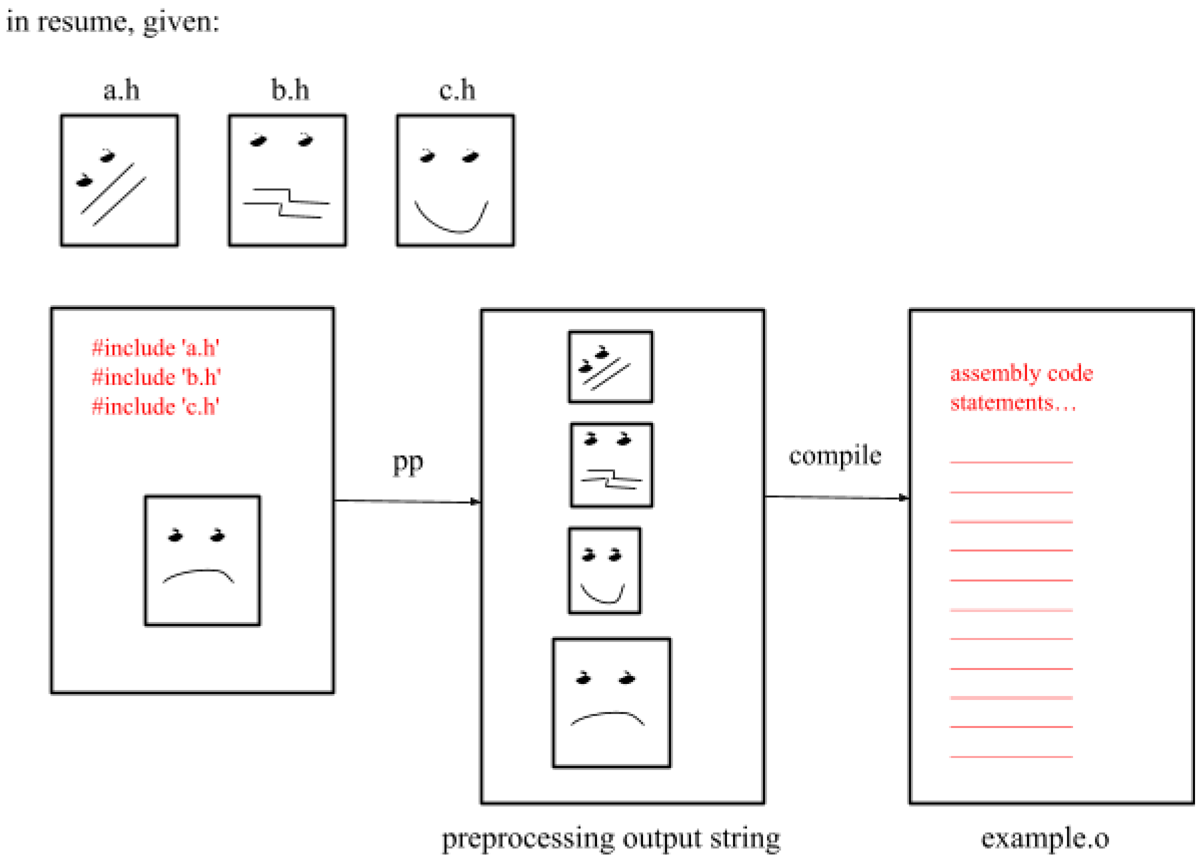
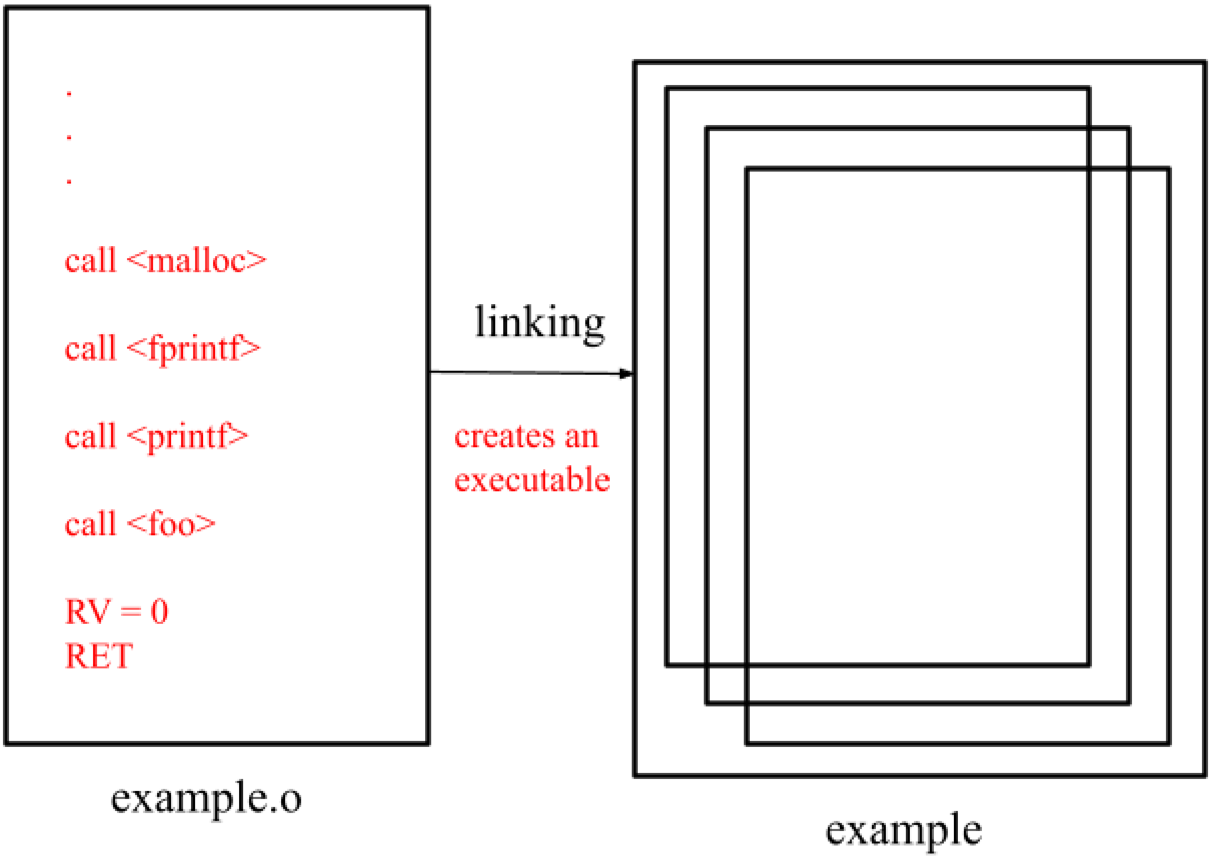
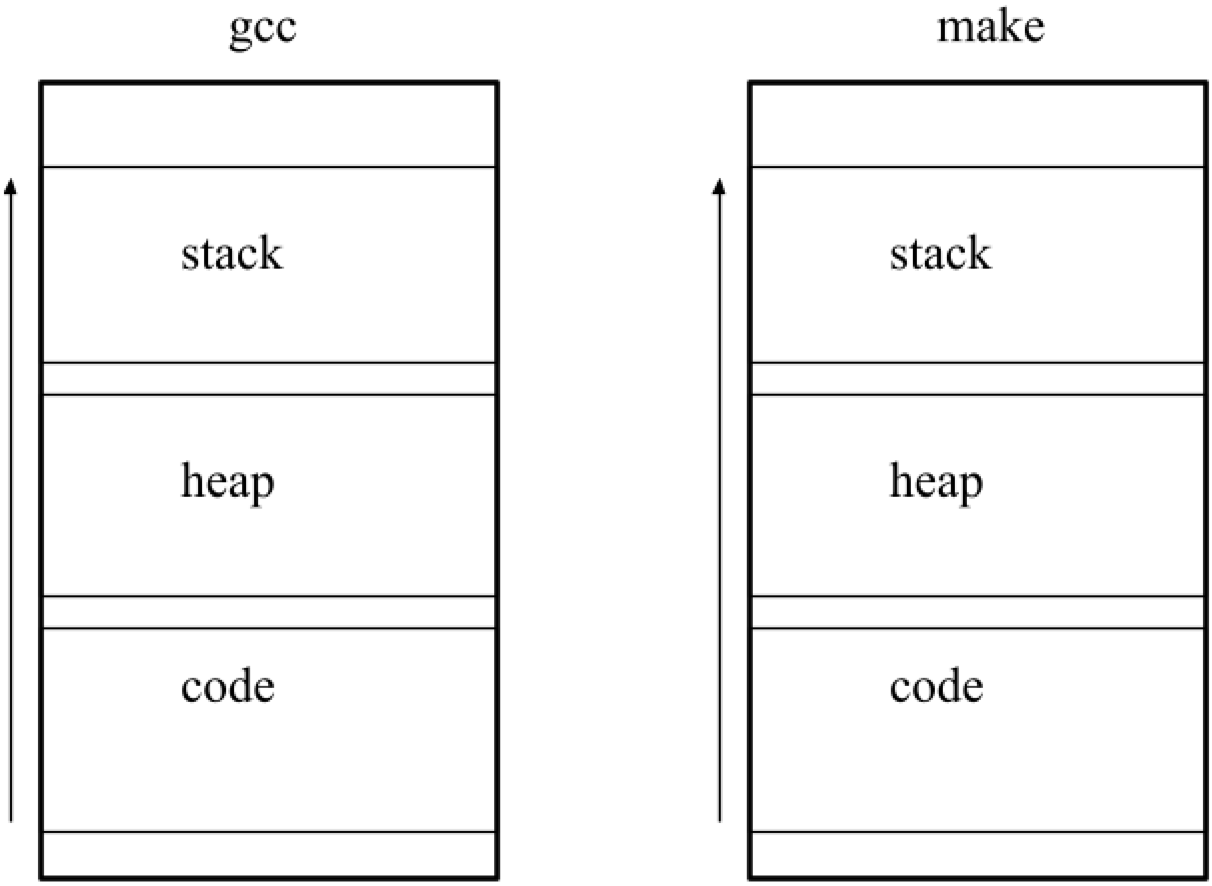
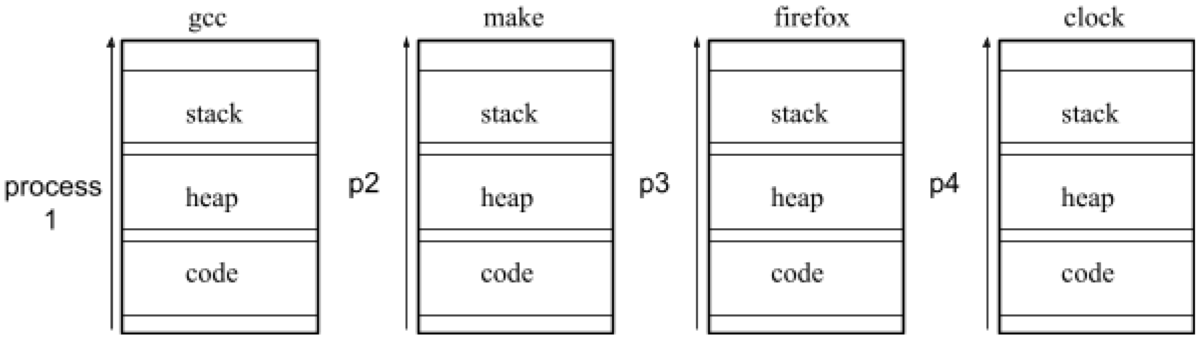
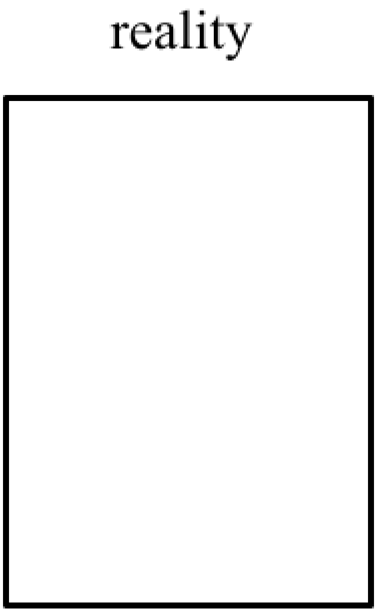
Another example could be the assert macro:

```
    #include <stdio.h>
    #include <stdlib.h>
    #define assert(cond) \
        ((cond) ? (void)0 : (fprintf(stderr, "Assertion failed:
    %s, file %s, line %d\n", #cond, __FILE__, __LINE__),
    exit(1)))

    int main() {
        assert(1 == 1);
        assert(1 == 2);
        return 0;
    }
```

```
>>> Assertion failed: 1 == 2, file test.c, line 8
```

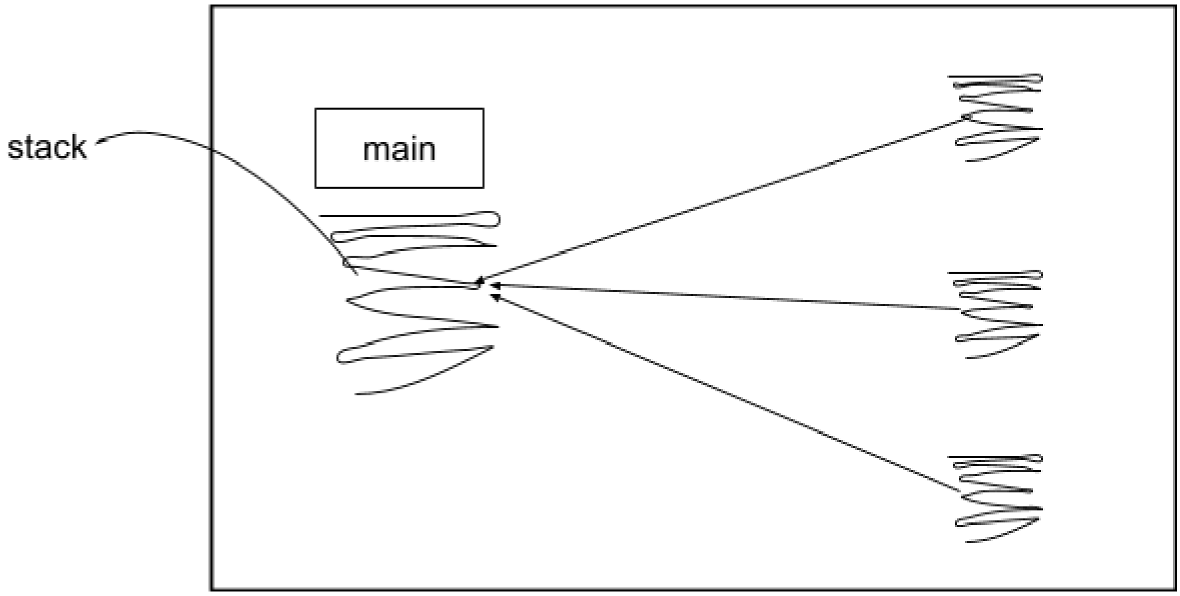
To see the immediate result of preprocessing you can use:

```
```

~~~~
      $ gcc -E file.c
```
~~~~

Compilation

As previously mentioned, after preprocessing, GCC processes a string of text that no longer includes #include or #define directives. The compiler takes this preprocessed file, which is free of directives, and translates it into assembly instructions, generating the .o file.

During compilation, the compiler checks for syntax errors, missing statements, and references to undefined variables and labels.



Linking

The linking phase is what turns your compiled object code (the .o files) into a single, cohesive program (an executable or library).

Symbol Resolution

Each `.o` file may contain references (calls or accesses) to code or data that live in other object files or libraries. For example, your object file might have instructions like `call <malloc>` or `call <foo>`—but does not itself contain the definition of these functions. The linker looks up all such undefined symbols in the other object files and in any libraries you've told it to use (e.g., the C standard library for `malloc, printf`, etc.) and then "resolves" those references.

Relocation

Once the linker knows where each symbol lives (which object file or library, and at what address), it modifies the machine instructions in your `.o` files so that function calls, variable accesses, and jumps go to the correct memory addresses in the final program. For instance, the placeholder `<malloc>` is replaced with the actual address (or offset) in the standard library where `malloc` resides.

Combining object files

After symbol resolution and relocation, the linker combines the various `.o` files into one final output—usually an executable (e.g., `a.out` on Unix-like systems) or a library (e.g., `.so` or `.dll`). It merges together sections like `.text` (machine code), `.data` (global variables), and other sections into a single file format that the operating system can load and run.

Optionally handling static and dynamic libs

- **Static Libraries** (`.a` on Unix, `.lib` on Windows) are essentially archives of object files. The linker extracts only the object code it needs from these archives.
- **Dynamic Libraries** (`.so` on Unix, `.dll` on Windows) are not merged in at link time; instead, references to external functions remain slightly more "indirect," and the OS loader finds the library at runtime.

Producing the Final Output

The result is an executable file (or a shared library), with all external references resolved, so that when you run the program, the calls to functions like `malloc` or `fprintf` correctly jump to the linked code in the standard library (or wherever else they're defined).



Note: Forgetting to include a header does not necessarily mean the code will fail at compilation or linking. During compilation, the compiler typically assumes prototypes (pre-C99) for functions without associated .h files. The input types and arity are inferred based on the first appearance of the symbol as provided by the client, and the return type defaults to int. At the linking stage, the linker searches for symbols corresponding to each function call within the available symbol tables provided by imported libraries and default libraries. Occasionally, even if a symbol lacks an included prototype, the compiler's inference may suffice, enabling the use of a default implementation of the function.

In summary, the linking phase merely seeks something that resolves the symbols in each function call and is entirely unaware of implementation-specific details.

Disclaimer: C++ provides better safety during linking compared to C because the assembly code generated in the `.o` file embeds function signatures in the symbol names (e.g `call <int_foo_int>`). This ensures that mismatched prototypes lead to linking errors, preventing potential runtime issues.

Concurrent programming

Memory delusion

Suppose you run `gcc` and `make` apps. Each will see a virtual memory address base like:



but these bases don't reside in the same place, apps generally don't share the `stack`, `heap` or `code`.

Now lets bring two more apps to the table and name a process to each:



Despite the fact that they all individually assume that their `stack`, `heap` and `code` segments are big enough to meet their needs, on a single processor machine there is only one address base (physical memory):



and this has to somehow host all of the memory that is meaningful to the processes that are running according to the code stored on `gcc`, `make`, `firefox`, and `clock` executables.

As it turns out, `gcc` and `make` may have the same virtual addresses for their respective `stack` segment, but physically they reside on a different place on the real base (they have different physical memory), and what the O.S will do to ensure that is to call the memory management unit to basically build a table of (`{process}`, `{address}`) that maps to a real address in the physical memory.

Multiprocessing and multithreading

Multiprocessing is a discipline on its own, and since it is essentially managed by the OS, we will focus more on threads which is where we have more control from within our app's implementation. But first let's quickly review the two:

What is multiprocessing?

Is when a system runs two or more processes simultaneously. Think of a **process** as a self-contained program with its own dedicated memory space. The operating system (OS) manages these processes and they can run in parallel on different CPU cores.

Key Characteristics:

- **Isolation:** Processes are isolated from each other. A crash in one process won't affect another;
- **Separate Memory:** Each process has its own private memory space;
- **Heavyweight:** Creating a new process is resource-intensive and takes more time;
- **Communication:** Communication between processes is complex and slower, managed by the OS.

What is multithreading?

Is the ability of a single process to manage multiple independent execution paths, known as **threads**. All threads within a process share the same memory space, including code and data.

Key Characteristics:

- **Shared Memory:** Threads within a process share the same memory and resources;
- **Lightweight:** Creating a thread is much faster and less resource-intensive than creating a process;
- **Efficient Communication:** Threads can communicate quickly and easily by reading from and writing to the same data structures;
- **Risk:** An error or crash in one thread can take down the entire process. Synchronization is crucial to prevent data corruption.

Moving on to a problem: suppose we have agents selling tickets out there, and we want to somehow simulate their selling behavior programmatically. We could start with something like:

```
int main() {
    int numAgents = 10;
    int numTickets = 150;

    for(int agent=1; agent<=numAgents; agent++) {
        sellTickets(agent, numTickets, numAgent);
        printf('agent %d All done\n', agent);
    }
    return 0;
}
```

having

```
void SellTickets(int agentId, int numTicketsToSell) {
    while(numTicketsToSell>0) {
        printf('agent %d sold a ticket\n', agentId);
        numTicketsToSell-;
    }
}
```

but that would be a very simple and unrealistic simulation since it presumes each agent will sell all of their tickets one after the other, synchronously. So we have to change it a little. Let's start modifying our main *for* loop and introduce Cain's lecture custom thread package *ThreadPackage*:

```
int main() {
    int numAgents = 10;
    int numTickets = 150;
    int ThreadPackage(False);

    for(int agent=1; agent<=numAgents; agent++) {
        char name[32];

        // this prints not to the console but to a character
buffer
        // in other words, rather than echoing the char o the
screen
        // we echo the chars in place as a char array and
make sure
        // that turns out to be a C string
        sprintf(name, 'agent %d thread\n', agent);
        ThreadNew(name, SellTickets, 3, agent,
numTickets/numAgents);
    }
```

```
        RunAllThreads();
        return 0;
    }
```

Well, but now we also need to update our SellTickets function to ensure every thread refers to the same global number of tickets to sell:

```
    void SellTickets(int agentId, int *numTicketsp) {
        while(*numTicketsp > 0) {
            printf('agent %d sold a ticket\n', agentId);
            (*numTicketsp)--;
        }
    }
```

Now every thread will refer to the same 150 tickets from the stack frame of the main thread



The problem now is that there is a window for a situation where threads commit to decrement the global reference (e.g passes **numTicketsp>0* condition) but their processing time slice ends immediately after that and the execution is handled to another thread that could as well commit to the same action as the counter hasn't been decremented yet, which in the end would compromise the integrity of our reference allowing it to become negative.

So, to solve that we have to somehow bind both condition and decrement altogether in a single/atomic execution step:

```
        (*numTicketsp > 0)
        (*numTicketsp)--;
```

```

For that we'll introduce a new data type from Cain's lecture library that is a nonnegative integer and supports atomic increments and decrements: *Semaphore*

Which will come along with some utility functions:

- *SemaphoreWait(Semaphore lock);*
- *SemaphoreSignal(Semaphore lock);*

But let's take a quick break before we talk about these functions to go through the analogy Jerry Cain brought to the class:

Imagine you are in a restaurant and feel an urge to go to the toilet, you reach the corridor and queue up, now it's your time to go and you walk through looking for an available cabin, you wiggle some doors to find the one that is not locked and eventually find one and use the toilet resources. If you forget to lock the door behind you when you go in then someone might wiggle the door and catch you on the fly which would be a bit inconvenient, right?

This is why we need `SemaphoreWait` and `SemaphoreSignal`

*SemaphoreWait* receives a *Semaphore* and decrements it atomically, if *Semaphore* is already zero by the time *SemaphoreWait* commits to decrement it then the calling thread does nothing and releases execution (as there is probably another thread waiting to increment Semaphore back with a *SemaphoreSignal* call). Think like reaching the toilet corridor when there is no queue but all cabins are full, you'll have to hold a bit until someone releases a cabin for you. And as you already figured, *SemaphoreSignal* releases a resource by incrementing *Semaphore*.

We can now make use of this and change our *ThreadNew* call at *main* a little bit:

````
```
    int main() {
        int numAgents = 10;
        int numTickets = 150;
        int ThreadPackage(False);

        // binary semaphore acting as a mutex
        Semaphore lock = SempahoreNew(-, 1);
        for(int agent=1; agent<=numAgents; agent++) {
            char name[32];
            sprintf(name, 'agent %d thread\n', agent);
            // The number `3` tells ThreadNew to look for the
    next three parameters and pass them as arguments to the
    SellTickets function.
            ThreadNew(name, SellTickets, 3, agent, &numTickets,
    lock);
        }
        RunAllThreads();
````

```
        return 0;
    }
```

Also our *SellTickets* func:

```
    void SellTickets(int agentId, int *numTicketsp, Semaphore lock) {
        while(True) {
            SemaphoreWait(lock);
            if(*numTicketsp==0) break;
            (*numTicketsp)--;
            printf('agent %d sold a ticket\n', agentId);
            SemaphoreSignal(lock);
        }
        SemaphoreSignal(lock); // safe net for also releasing the lock whenever the loop breaks
    }
```

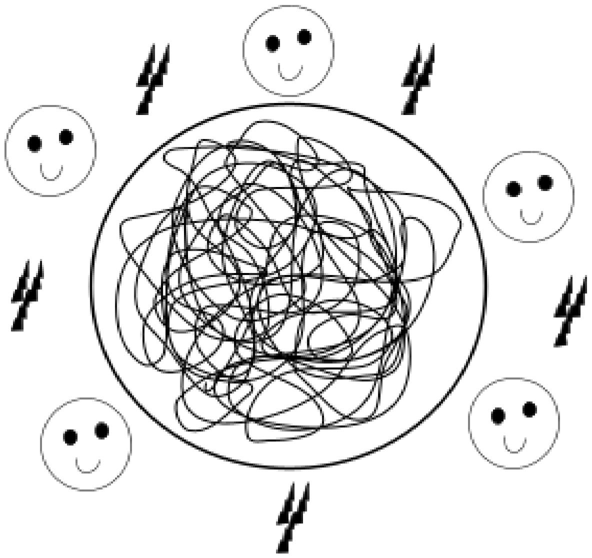
Note: we need to be very careful here, not only with the implementation sequence but also the initialization, for instance if for some reason you accidentally initialize *Semaphore* as 0 instead of 1 in this particular scenario, then all threads will get stuck into the loop, putting the program in a state known as ***dead lock***.

The dining philosophers problem

https://en.wikipedia.org/wiki/Dining_philosophers_problem

Problem statement

Every philosopher follows the same formula every single day, they wake up, they think for a while and they eat, then think for a while, eat, think for a while, eat and lastly think for a while before they go to bed, in summary there are four think sessions interleaved by three eat sessions. The problem is that in order to eat, a philosopher has to grab both forks on either side of them and at least one of the forks is shared.



At first we could naively implement the following to simulate the problem:

```
Semaphore forks[] = {1, 1, 1, 1, 1}

void Philosophers(int id)
    {
    for(int i=0; i<3; i++) {
        Think();
        SemaphoreWait(forks[id]);
        SemaphoreWait(forks[(id+1)%5]);
        Eat();
        SemaphoreSignal(forks[id]);
        SemaphoreSignal(forks[(id+1)%5]);
    }
}
```

But that could lead to a very specific kind of deadlock, see: because thread execution gets pulled out of the processor every time the first fork is grabbed, this can leave the dinner table in a state where every one has at least one fork in hand and there are no more forks left.

There are more than one heuristic capable of solving that problem, for instance, we could add some logic that only allows for odd indexed forks to be grabbed first. But we'll stick to adding an extra *SemaphoreWait* and *SemaphoreSignal* to *numAllowedToEat*, that will ensure only a maximum of 4 philosophers can eat simultaneously. This leaves the thread manager with maximum flexibility without deadlock. Take some time to look at the code below and understand why it works.

```
Semaphore forks[] = {1, 1, 1, 1, 1}
Semaphore numAllowedToEat(4);
```

```
void Philosophers(int id)
    {
    for(int i=0; i<3; i++) {
        Think();

        SemaphoreWait(numAllowedToEat);

        SemaphoreWait(forks[id]);
        SemaphoreWait(forks[(id+1)%5]);

        Eat();

        SemaphoreSignal(forks[id]);
        SemaphoreSignal(forks[(id+1)%5]);

        SemaphoreSignal(numAllowedToEat);
    }
}
```

Moving on to a more complex example

The Ice Cream Store Simulation

Actors:

clerks → make ice cream cones
manager → supervises work of clerks
customer → buy cones
cashier → rings up each customer

Note: different threads are launched for each of the actors.

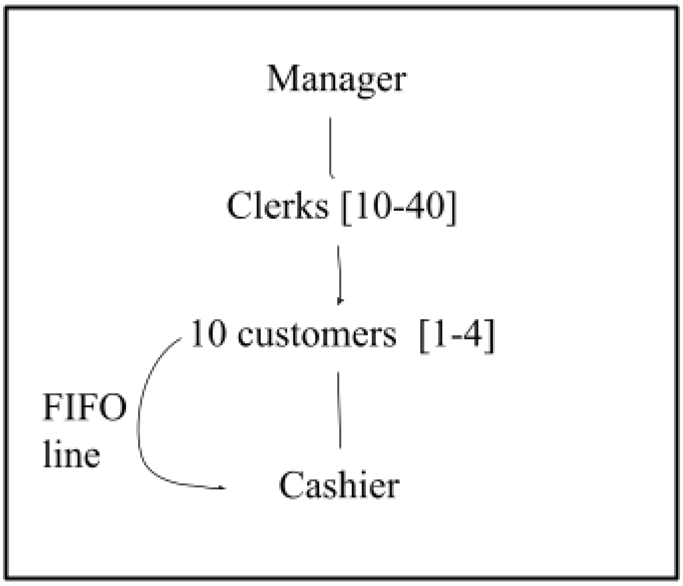
Problem statement

Each customer wants to order a few ice cream cones, wait for them to be made, then get in line to pay the cashier, and leave after paying. The customers are impatient and don't wait for one clerk to make several cones, instead the customers dispatch one clerk thread independently for each cone. Once the customer has gotten their cones made, they get in line for the cashier and wait for their turn. After paying the cashier, the customer leaves.

Each clerk makes exactly one ice cream cone, each cone is verified by the manager, if it is not ok it is thrown away and the clerk has to make another. Once a cone passes QA the clerk hands the cone to the customer and is done.

The manager sits idle until a clerk needs a cone inspected. When it gets an inspection request, the inspector determines if it passes and lets the clerk know how the cone fared. The manager is done when all cones have been inspected.

The line for the cashier is maintained in FIFO order. The cashier sits idle while there are no customers in line. When a customer is ready to pay, the cashier handles the bill. Once the bill is paid, the customer can leave. Once all customers have paid, the cashier is finished.



Take a moment to think about the problem and then try to implement each actor and its respective loops and semaphores, don't worry too much about the language here, focus on the signaling and waiting mechanics, then when you think you are done check out the original solution to the problem to see how good you went: https://see.stanford.edu/materials/icsppcs107/26-More-Concurrency.pdf