COMP281
Principles of C and memory management

lecture 2

Dr. Frans Oliehoek
Department of Computer Science,
University of Liverpool.

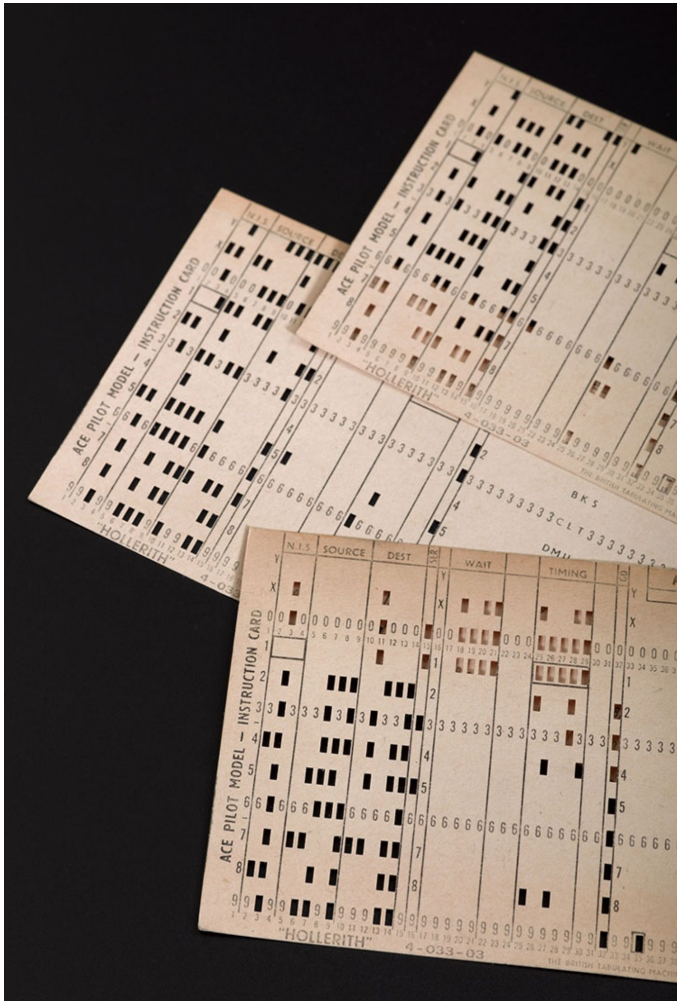
Today

- Some more historical context of C
- Memory organization: Stack, Heap and all that…
- Some more C language element

Some more historical context

Programming

- Early programs had to be written by manipulating the bits directly on punch cards. (e.g, there are 3 bits to identify the ‘destination’ of each instruction)
- To make this easier – ‘Assembly language’ programming was introduced.
- This uses ‘mnemonics’ to describe what the hardware will do
- An ‘assembler’ program also assists – such as by counting memory locations automatically
- Most of you have seen this in COMP103



Programming languages

- **High-level languages**, such as ALGOL, Fortran and Lisp were invented in the 1950s that *abstracted* away from the hardware and would allow the same program to run on many different computer systems.
  - Each of these languages was good to represent certain types of problem.

- During the 60s, 70s, 80s, and even 90s, computers were not nearly as powerful as they are today – and **many programs would still be written in assembly language**
  - (e.g., almost all computer games before 1990).

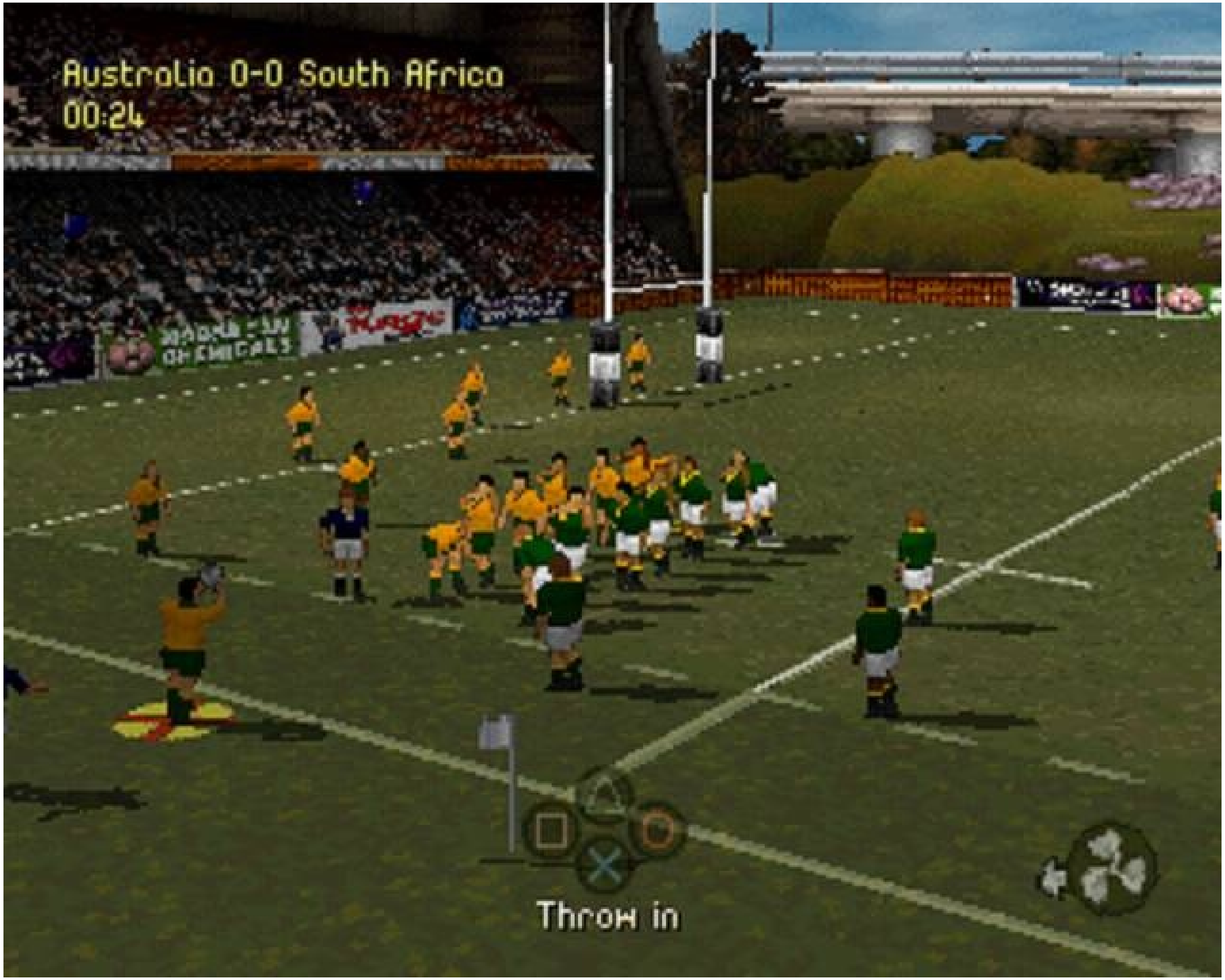
- There was a need to write **complex programs that could run efficiently** on many different types of hardware.... Hence the invention of C

A Super Nintendo had a **4mhz** CPU…

The C Language

- By 1973 the C language had become powerful enough that most of the Unix kernel was written in C
- Unix was one of the first operating system kernels implemented in a language other than assembly.
- By 1993 - 1994, **computer games systems** (such as 3DO, Playstation, Saturn) were getting more complex and were provided with an **operating system**
- **Most games development switched to C**, as it was quite efficient, easier to write, allowed easier access to the hardware and could be shared across multiple different systems.
- The **ANSI C standard** was first approved in 1989. Later versions were approved by the ISO in 1999 and 2011.

A Playstation (1) had a 33mhz CPU…



IBM 7090 Programming in the 1950's

```
ENTRY   SXA   4,RETURN
        LDQ   X
        FMP   A
        FAD   B
        XCA
        FMP   X
        FAD   C
        STO   RESULT
RETURN  TRA   0
A       BSS   1
B       BSS   1
C       BSS   1
X       BSS   1
TEMP    BSS   1
STORE   BSS   1
        END
```

IBM 7090 Programming in the 1950's

```
ENTRY     SXA     4,RETURN
          LDQ     X
          FMP     A
          FAD     B
          XCA
          FMP     X
          FAD     C
          STO     RESULT
RETURN    TRA     0
A         BSS     1
B         BSS     1
C         BSS     1
X         BSS     1
TEMP      BSS     1
STORE     BSS     1
          END
```

if the following was nearly as efficient… what would you prefer?

```
//C hello world example
#include <stdio.h>

int main()
{
  printf("Hello world\n");
  return 0;
}
```

About Memory

Assembly language Example

```
org 100h
main:
     mov dx,get_message
     call dx
     mov ah, 9
     int 21h
;this is a BIOS call to print out text
     mov ah, 0
     int 16h;  wait for a key

     mov ax, 4c00h
     int 21h
get_message:
     mov dx,msg
     ret
     resb 6
msg: db "Testing the program"
     db 10
     db  "Press any key$"
; the $ marks the end of the text
```

- Note that labels, denoted by a : may refer to the program or the data.
- A ***register***, such as dx, just contains a (16-bit) number.
- The processor does not care if it is just a number or an 'address'
- The address of a label is simply an indication of where it is in memory.
- The processor can also treat an address as either program code or data – they are both just locations in the main memory.

Assembly language Example

```
org 100h
main:
    mov dx,get_message
    call dx
    mov ah, 9
    int 21h
;this is a BIOS call to print out text
    mov ah, 0
    int 16h;  wait for a key

    mov ax, 4c00h
    int 21h
get_message:
    mov dx,msg
    ret
    resb 6
msg: db "Testing the program"
    db 10
    db  "Press any key$"
; the $ marks the end of the text
```

- Note that labels, denoted by a : may refer to the program or the data.
- A ***register***, such as dx, just contains a (16-bit) number.
- The processor does not care if it is just a number or an 'address'
- The address of a label is simply an indication of where it is in memory.
- The processor can also treat an address as either program code or data – they are both just locations in the main memory.

Assembly language Example

```
org 100h
main:
    mov dx,get_message
    call dx
    mov ah, 9
    int 21h
;this is a BIOS call to print out text
    mov ah, 0
    int 16h;  wait for a key

    mov ax, 4c00h
    int 21h
get_message:
    mov dx,msg
    ret
    resb 6
msg: db "Testing the program"
    db 10
    db  "Press any key$"
; the $ marks the end of the text
```

Address	**Data**	**Disassembly**
00000100	**BA1201**	**mov dx,0x112**
00000103	**FFD2**	**call dx**
00000105	**B409**	**mov ah,0x9**
00000107	**CD21**	**int 0x21**
00000109	**B400**	**mov ah,0x0**
0000010B	**CD16**	**int 0x16**
0000010D	**B8004C**	**mov ax,0x4c00**
00000110	**CD21**	**int 0x21**
00000112	**BA1C01**	**mov dx,0x11c**
00000115	**C3**	**ret**
00000116	0000	add [bx+si],al
00000118	0000	add [bx+si],al
0000011A	0000	add [bx+si],al
0000011C	54	push sp
0000011D	657374	gs jnc 0x94
...		

Assembly language Example

```
org 100h
main:
     mov dx,get_message
     call dx
     mov ah, 9
     int 21h

     mov ah, 0
     int 16h;  wait for a key

     mov ax, 4c00h
     int 21h
get_message:
     mov dx,msg
     ret
     resb 6
msg: db "Testing the program"
     db 10
     db  "Press any key$"
; the $ marks the end of the text
```

Address	Data	Disassembly
00000100	BA1201	mov dx,0x112
00000103	FFD2	call dx
00000105	B409	mov ah,0x9
00000107	CD21	int 0x21
00000109	B400	mov ah,0x0
0000010B	CD16	int 0x16
0000010D	B8004C	mov ax,0x4c00
00000110	CD21	int 0x21
00000112	BA1C01	mov dx,0x11c
00000115	C3	ret
00000116	0000	add [bx+si],al
00000118	0000	add [bx+si],al
0000011A	0000	add [bx+si],al
0000011C	54	push sp
0000011D	657374	gs jnc 0x94
...		

Takeaway: It is all just addresses and instructions...

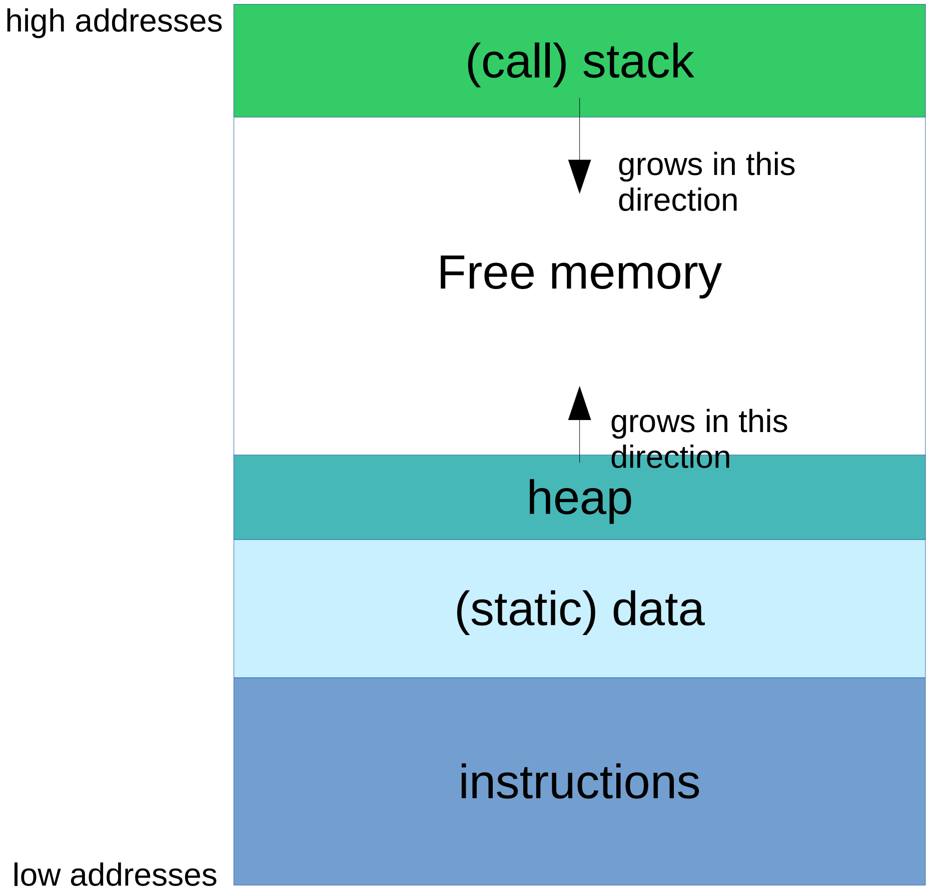
how to make some sense out of this...?

- where to find data?
- where to return to after 'function'?

→ Organize in meaningful way

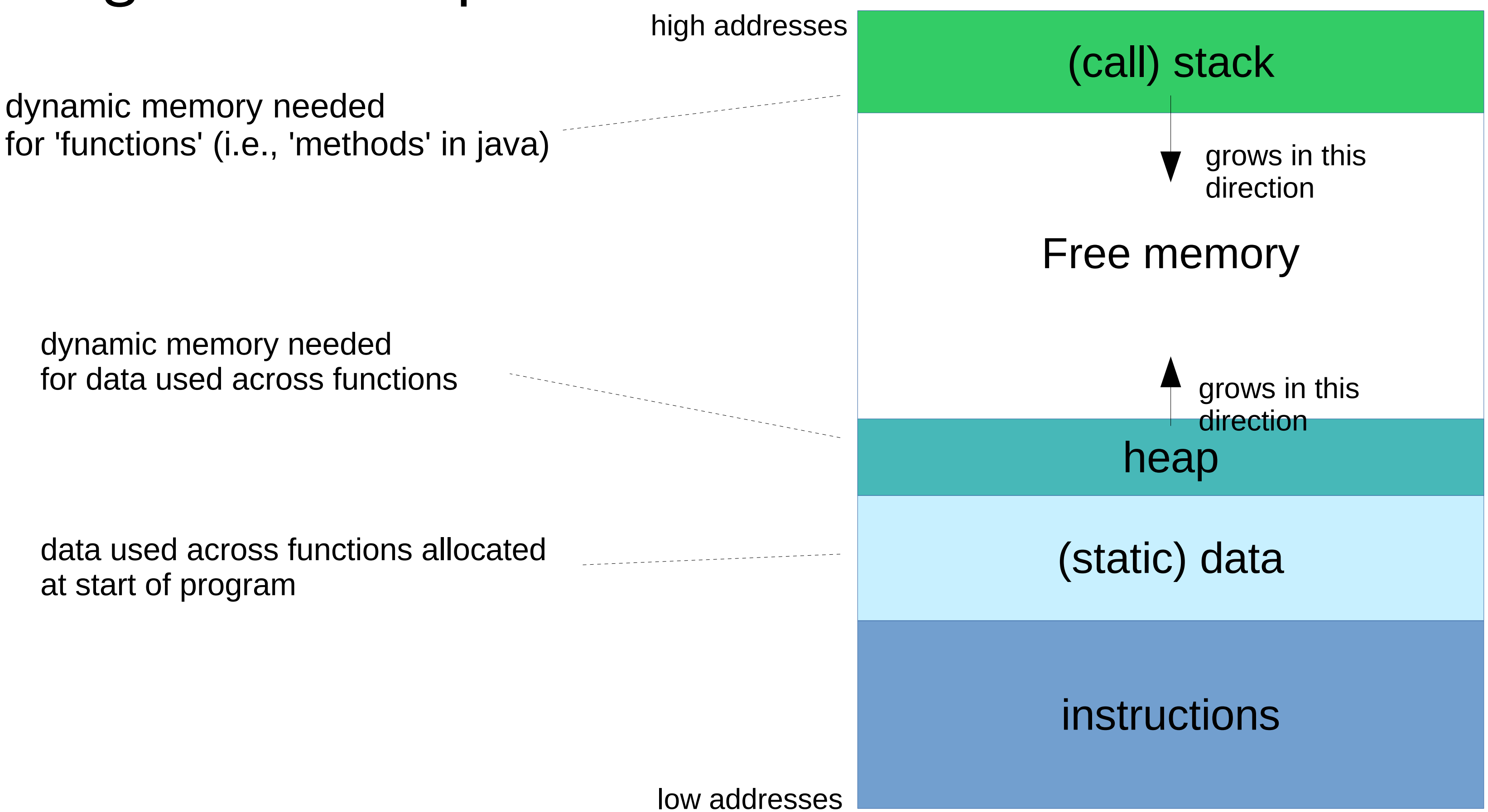
How memory is organized

- Organization depicted:



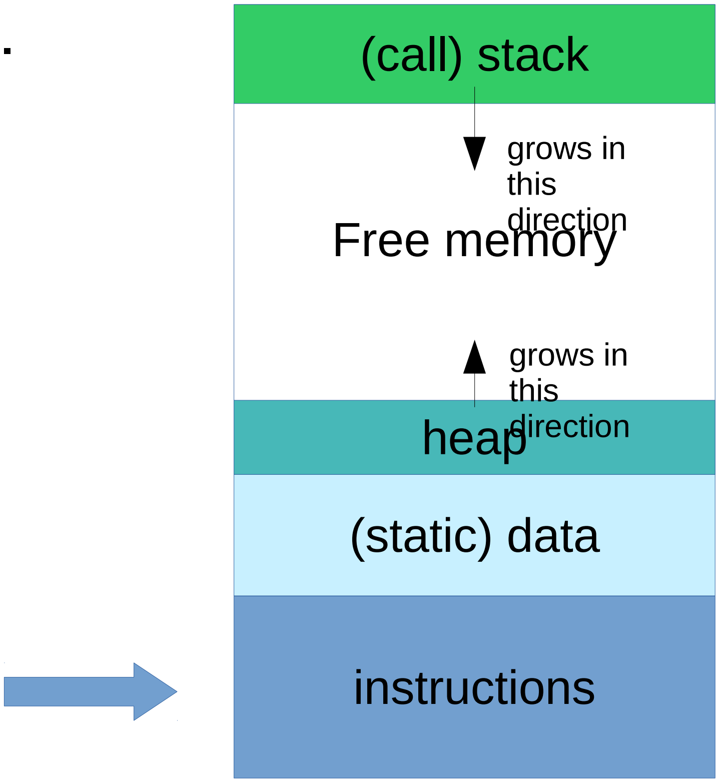
How memory is organized

- Organization depicted:



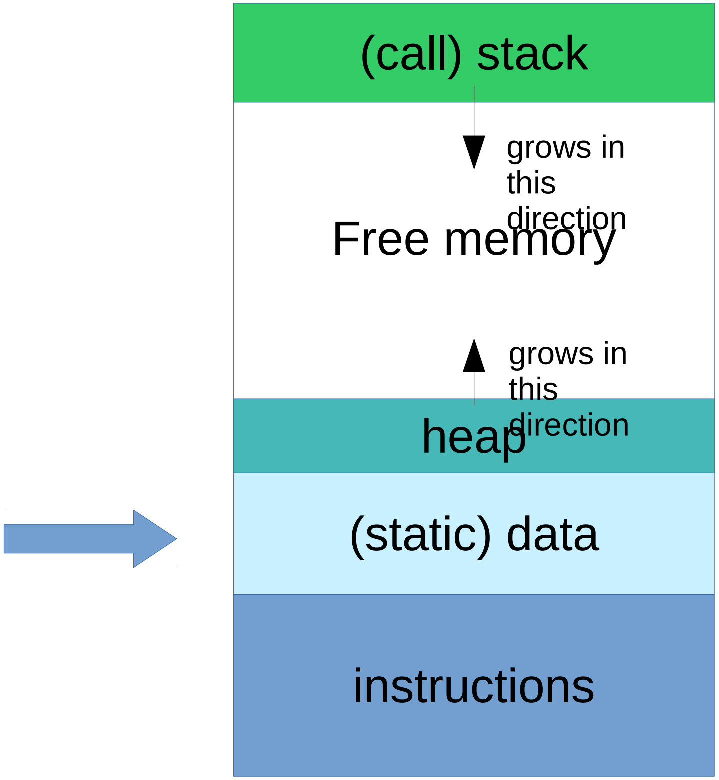
Instructions ('code segment')

- Compiler produces instructions and they are load into this part of memory.
- You need not worry about this…



(Static) data segment

- Will contain data that exists throughout the program
- later this lecture…

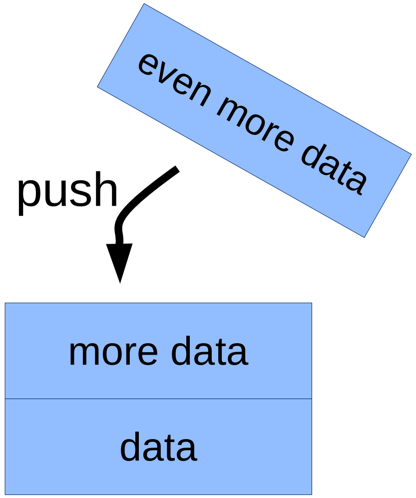


The Stack (also "Call Stack")

- You saw a 'call' instruction and a 'ret' instruction…
  - Q. How does the processor know where to get back to?
  - Temporary store the return address (as well as other things…) on **the stack**

- Stack??
  - written to with a 'push' instruction
  - read with a 'pop' instruction

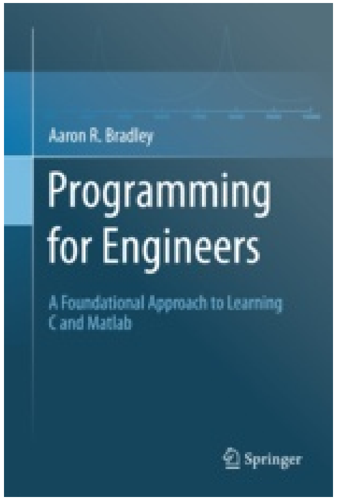
- Calling a function instruction
  - **pushes** a return address onto the stack
  - then 'jumps' to the required address
- the return instruction
  - pops the return address off the stack
  - then jumps to it



More on the Stack

- We will cover more details later...
- But I can strongly recommend Chapter 1 of this book:

Programming for Engineers:
A Foundational Approach to
Learning C and Matlab
Authors: Bradley, Aaron R.

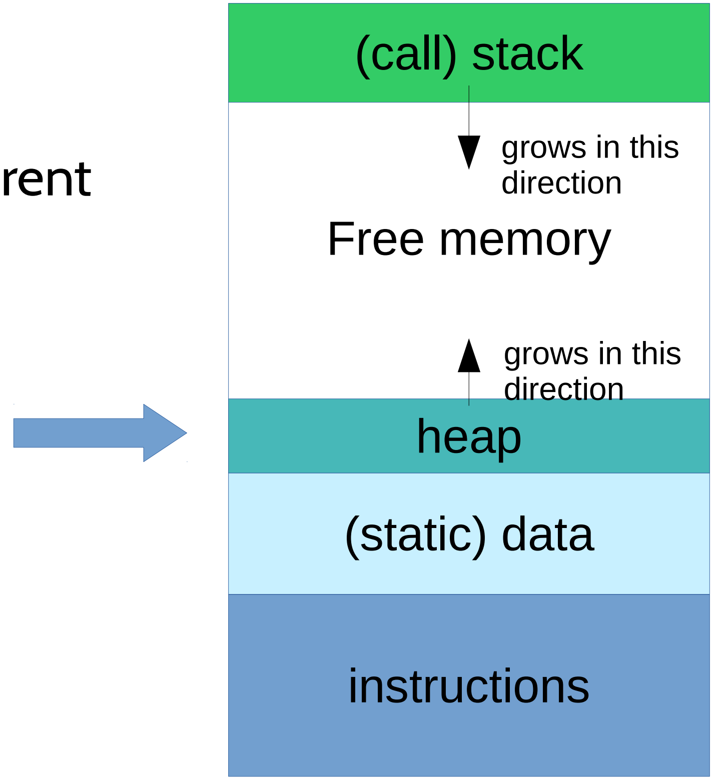


- http://link.springer.com/978-3-642-23303-6

Heap

- Other forms of **dynamic** memory allocation: in 'the heap'
  - Not connected to functions or function calls
  - so for data that is persistent across different function calls

- Will see this in later lectures...



First words about memory – review

- Why is memory management important?

- Processor just uses binary 'data'
  - it does not care if this is a value, or an address, or what type of address (value or instruction)

- Assembly language exposes the full range of possibilities
  - and C is very close in this respect.
  - in contrast to Java (and many other languages)

- However, C does give *some* structure to memory usage:
  - code segment, static data, stack and heap.

Some more C language elements

Functions: declaration, definition

- A function is the same as a Java method
  - Except it has no Object to work with, so no `this` variable
- a block of code that (optionally) takes some input and returns some value as output

```
int add(int a, int b);
int v1=42;

int main()
{
    int v2=3;
    int result;

    result = add(v1, v2);
    printf("result = %d", result);
    return(0);
}

int add(int a, int b)
{
    return a+b;
}
```

Functions: declaration, definition

- A function is the same as a Java method
  - Except it has no Object to work with, so no `this` variable
- a block of code that (optionally) takes some input and returns some value as output

```
int add(int a, int b);
int v1=42;

int main()
{
    int v2=3;
    int result;

    result = add(v1, v2);
    printf("result = %d", result);
    return(0);
}

int add(int a, int b)
{
    return a+b;
}
```

declaration – such that compiler knows the function exists.

Functions: declaration, definition

- A function is the same as a Java method
  - Except it has no Object to work with, so no *`this`* variable
- a block of code that (optionally) takes some input and returns some value as output

```
int add(int a, int b);
int v1=42;

int main()
{
    int v2=3;
    int result;

    result = add(v1, v2);
    printf("result = %d", result);
    return(0);
}

int add(int a, int b)
{
    return a+b;
}
```

declaration – such that compiler knows the function exists.

definition – tells *how* it works

Functions: declaration, definition

- A function is the same as a Java method
  - Except it has no Object to work with, so no *this* variable
- a block of code that (op … output

```
int add(int a, int b)
int v1=42;

int main()
{
    int v2=3;
    int result;

    result = add(v1, v2);
    printf("result = %d", result);
    return(0);
}

int add(int a, int b)
{
    return a+b;
}
```

declaration – such that compiler knows the function exists

definition – tells *how* it works

That seems complicated... what is the use?
→ useful when splitting a project over files!

- **header files** (e.g., calculations.h)
  - will contain declarations
  - can be included by other files
- **implementation files** (e.g., calculations.c)
  - will contain definitions
  - need to be compiled just once, even if functionality used in multiple other files

Functions: no overloading

- Unlike Java, functions may not have the same name, even when the parameters are different (i.e., no *overloading*)
  - So you may need to do something like

```
int add_int(int a, int b)
{
    return a+b;
}

double add_double(double a, double b)
{
    return a+b;
}
```

Functions: use of variables

- Can use different types of variables

```
int add(int a, int b);
int v1=42;
```

global variable – usable by all functions

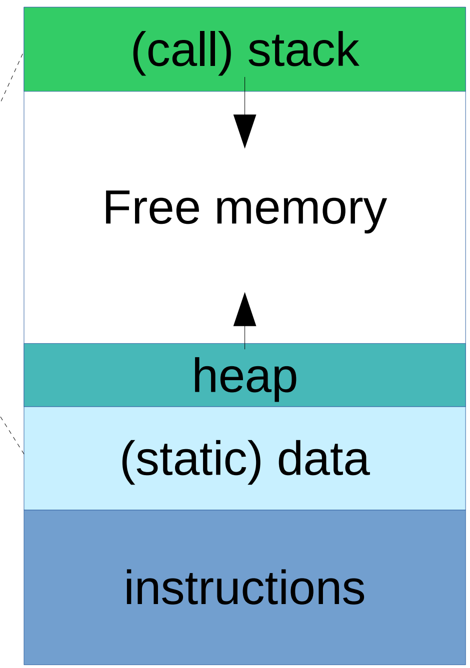
```
int main()
{
    int v2=3;
    int result;

    result = add(v1, v2);
    printf("result = %d", result);
    return(0);
}
int add(int a, int b)
{
    int res = a+b;
    return res;
}
```

local variables – usable by only 1 function

'Global' and 'Local' Variables

- **Global variables** are any that are declared outside a function.
  - Stored in static data segment: they take up space for the entire time the program is running.
  - They keep their value through the life of the program
  - The space is allocated in main memory, and they are initialised to 0 by default.
  - They can be used to have functions communicate… …but this is often considered 'evil'



- **Local variables** are allocated space on the stack
  - Every time the function is called, the space is allocated again
  - So they don't keep the same value in-between calls
  - They are not initialised by default
  - They could start with any value

'Static' local variables

- May make sense to want a global variable that is only accessed by a single function.
  - i.e., a variable that is used in a single function, but which needs to retain its value…
- This is possible: declare the local variable to be **static**
  - The ‘extent’ is global, but the ‘scope’ is local
  - they take up space and retain their value for as long as the program runs
  - But they cannot be accessed from any other function
  - they are initialised to 0 by default

'Static' local variables

- May make sense to want a global variable that is only accessed by a single function.
  - i.e., a variable that is used in a single function, but which needs to retain its value..
- This is possible: declare the local variable to be static
  - The 'extent' is global, but the 'scope' is local
  - they take up space and retain their value for as long as the program runs
  - But they cannot be accessed from any other function
  - they are initialised to 0 by default

For instance:

```
void senseless_function()
{
  static int count = 0;
  printf("Called %d times now", ++count);
}
```

More Methods of Input / EOF

- You can also read a limited number of characters with `fgets`
- `stdin` is the standard (usually keyboad) input, like System.in in Java

```
main()
{   char input[10];
    fgets(input, 10, stdin);
    printf(input);
}                                   file:e5.c
```

- You can also read characters individually

```
int c;
c = getchar();
if (c == EOF)
{ .. process End of File .. }
```

- You can also check the return value of `fgets` – NULL indicates that the end of the input was reached

More Methods of Input / EOF

- You can also read a limited number of characters with `fgets`
- `stdin` is the standard (usually keyboad) input, like System.in in Java

```
main()
{   char input[10];
    fgets(input, 10, stdin);
    printf(input);

}                                       file:e5.c
```

- You can also read characters individually

```
int c;
c = getchar();
if (c == EOF)
{ .. process End of File .. }
```

'EOF' is the special symbol indicating the end of input

- You can also check the return value of `fgets` – NULL indicates that the end of the input was reached

Arrays

- Like Java, an array is a randomly-accessible list of the same type
  - elements may be accessed, like Java, by using square brackets
  - valid indices: `0, ..., (num_elements-1)`
- It is stored as **consecutive blocks** of memory
- Example:

```
int main(void)
{
  int table[12];
  int i;
  for (i=0; i < 12; i++)
    table[i] = i * 12;

  for (i=0; i < 12; i++)
    printf("%d \t %d \t%d\n",table[i]);
}
```

Arrays will not catch any errors…

- You are responsible for getting indexing right!

```
int a[1];
int b[1];
int c[1];

main()
{
  a[0]=0;
  b[0]=0;
  c[0]=0;

  b[1] = 42;
  b[-1]= 3;

  printf("%d %d %d\n", a[0],b[0],c[0]);
}
```

File:q9.c

```
main()
{
  int a[1];
  int b[1];
  int c[1];

  a[0]=0;
  b[0]=0;
  c[0]=0;

  b[1] = 42;
  b[-1]= 3;

  printf("%d %d %d\n", a[0],b[0],c[0]);
}
```

File:q10.c

Arrays will not catch any errors…

- You are responsible for getting indexing right!

```
int a[1];
int b[1];
int c[1];

main()
{
  a[0]=0;
  b[0]=0;
  c[0]=0;

  b[1] = 42;
  b[-1]= 3;

  printf("%d %d %d\n", a[0],b[0],c[0]);
}
```

File:q9.c

Behavior unpredictable… depends on many aspects (compiler, OS, where data stored, etc.)

```
main()
{
  int a[1];
  int b[1];
  int c[1];

  a[0]=0;
  b[0]=0;
  c[0]=0;

  b[1] = 42;
  b[-1]= 3;

  printf("%d %d %d\n", a[0],b[0],c[0]);
}
```

File:q10.c

Arrays on the stack...

- As for other variables, arrays may be **local** or **global**
- What could be the problem with the following?

```
main()
{
    int array[10000000];
    int i =0;
    for (i=0; i < 10000000;i++)
        printf("%d ",array[i]);
}
```

stack_size.c

Arrays: Strings

- strings use a `char` array
- The standard libraries assume that it ends with a ‘0’ (written `'\0'`)

```
#include<stdio.h>
#include<string.h>
int main (void)
{
    char text[] = "This is a string";
    char text2[] = "This is \0 a string";

    printf ("%d %d \n %s \n %s \n",
      strlen(text), strlen(text2), text, text2
    );

    int length = strlen (text) + 1;
    int i = 0;

    for (i = 0; i < length; i++)
        printf ("%d ", text[i]);
}
```

example_string.c

Arrays: Strings

- strings use a `char` array
- The standard libraries assume that it ends with a '0' (written '`\0`')

```
#include<stdio.h>
#include<string.h>
int main (void)
{
    char text[] = "This is a string";
    char text2[] = "This is \0 a string";

    printf ("%d %d \n %s \n %s \n",
      strlen(text), strlen(text2), text, text2
    );

    int length = strlen (text) + 1;
    int i = 0;

    for (i = 0; i < length; i++)
        printf ("%d ", text[i]);
}
```

```
$ ./a.out
16 8
 This is a string
 This is
84 104 105 115 32 105 115 32 97 32 115
116 114 105 110 103 0
```

`example_string.c`

Strings

- How does `strlen` find the length of a string?
  - By counting from the beginning until it find '\0'

- Other string functions
  - `strcmp` - compare two strings, it returns
    - negative if the first string is lower,
    - 0 if they are the same,
    - positive if the second string is lower
  - `strcoll` - same, but takes into account character encoding (e.g. ä < b)
  - `strcpy` - copy a string
  - `strcat` - concatenate two strings

- don't forget to use `#include<string.h>`

Summary of C elements

- functions: declarations & definitions
  - for multi-file projects
- global vs local variables
  - “static local” local variables
- more input: fgets, getc, EOF
- arrays
- strings

Review

- More motivation for C
  - it is fast and essentially allows you to do pretty much everything that assembly code would
- A first view of the memory organization employed in C.
  - functions store local variables and return addresses on 'stack'
  - global variables stored in 'static data segment'
- Some more C language elements
  - functions,
  - syntax global, local variables,
  - more input
  - arrays, strings

suggested readings

- operators: K&R Ch2 (quick scan)
- input/output K&R Ch7 / Bradley Ch5
- stack: Bradley Ch1 / Lu Ch2

and possibly:

- control flow: Bradley Ch2 / K&R Ch 3
- functions: K&R Ch4