Quiz(20%)

10.19 Thursday

13:00pm--14:00pm

No workshop this week

Unit 8:
Regions of Memory:
Global, and Local Space

Hexadecimal

Binary / Decimal

- Assume a memory cell consists of 8 bits (one byte).
- We could always give the value of those 8 bits when discussing memory contents. (painful)
- We could convert them into decimal. (also painful)
  - E.g. $11111001_b = 249_d$

Hexadecimal : numbers to the base 16

binary 1 1 1 1 1 0 0 1

decimal 15 9

Hex F 9

Two Hex digits represent the value contained in a single byte (8 bits).

decimal	hexadecimal
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

- Consider an unsigned 16-bit integer.
- In Decimal, the conversion is even worse than for 1 byte.

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
1	1	0	1	0	1	1	1	1	0	1	1	1	0	0	1

$$32768+16384+4096+1024+512+256+128+32+16+8+1$$
$$= 55225$$

Hex

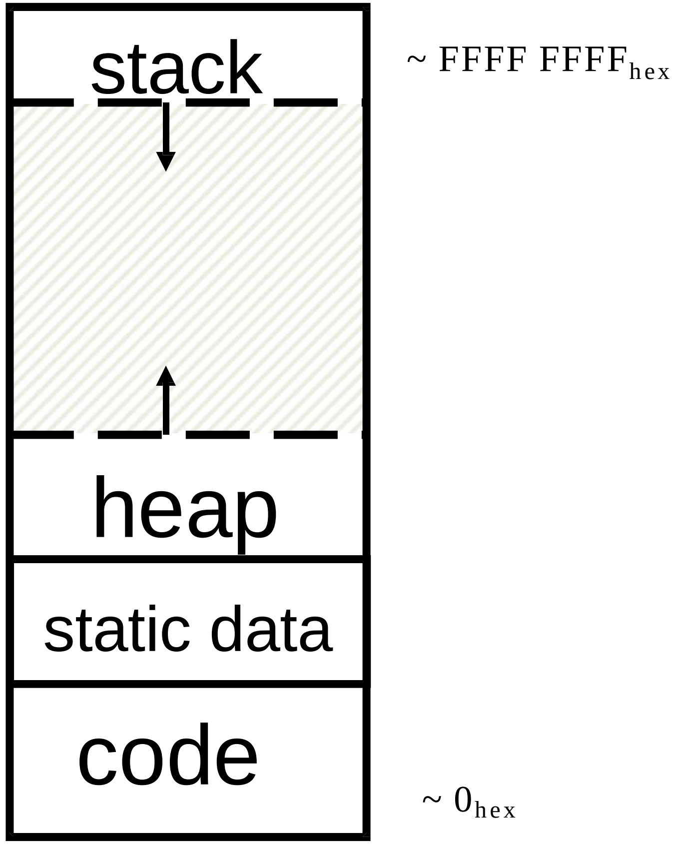
- With hexadecimal it is MUCH easier.

8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1
1	1	0	1	0	1	1	1	1	0	1	1	1	0	0	1
D				7				B				9			

Memory Space

Typical C Memory Management

- stack
  - local variables, parameters, return address
  - grows downward
- heap
  - space requested via e.g. `malloc()` ;
  - resizes dynamically, grows upward
  - data lives until deallocated by programmer
- static data
  - variables declared outside any functions, does not grow or shrink i.e. globals
- code
  - loaded when program starts, normally does not change



STATIC DATA REGION

Local Variables in 'C'


```
int age = 0x7; // hex
char name [7] = "louise";
int salary = 0x7654; // hex

void doIncrement()
{
        int increment = 0x10; // hex
        salary = salary + increment;
}

void main()
{
        char reverse [7];
        doIncrement();
        printf("salary plus increment = %d\n",salary);
}
```

"increment" is local to the body of "doIncrement". This means it can only be referred to within "doIncrement".

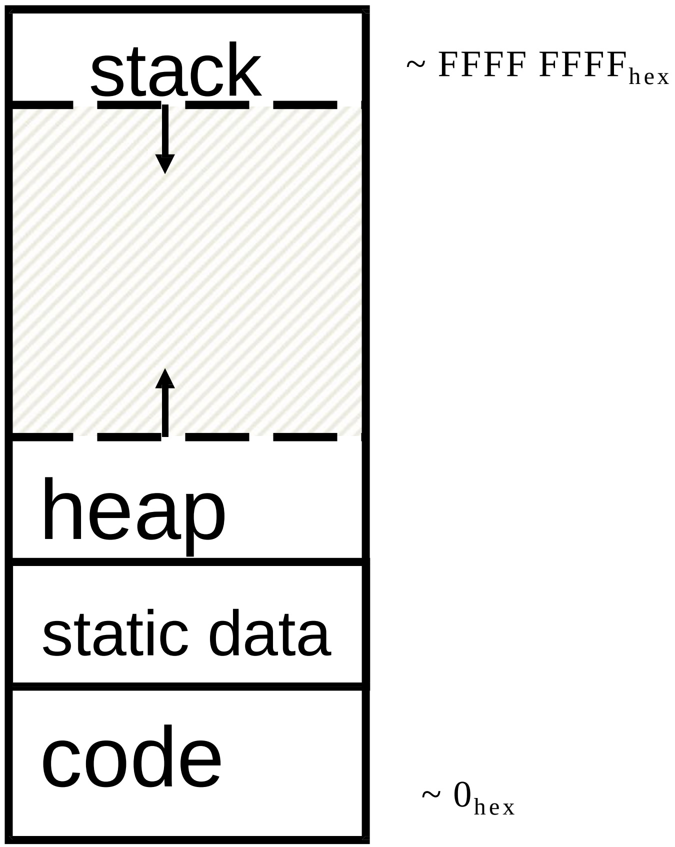
If in the "main" procedure we referred to "increment", this would result in a compile-time error.

Global Variables in ‘C’


```
int age = 0x7; // hex
char name [7] = “louise”;
int salary = 0x7654; // hex

void doIncrement(){
        int increment = 0x10; // hex
        salary = salary + increment;
}

void main() {
        char reverse [7];
        doIncrement();
        printf(“salary plus increment = %d\n”,salary);
}
```

Quiz 12

```
int  y = 0;
int  array[10];
int *p;

int main(){
        int x;
        int *pt = NULL;
        p=malloc(sizeof(x));
        return 0;
}
```



Memory allocation for global variables

- Assumptions:
  - The target machine has:
    - A 32-bit word size (i.e. the machine can load/store 4 bytes with one operation)
    - Memory is addressed on a per-byte basis

Some globals

- The globals ….

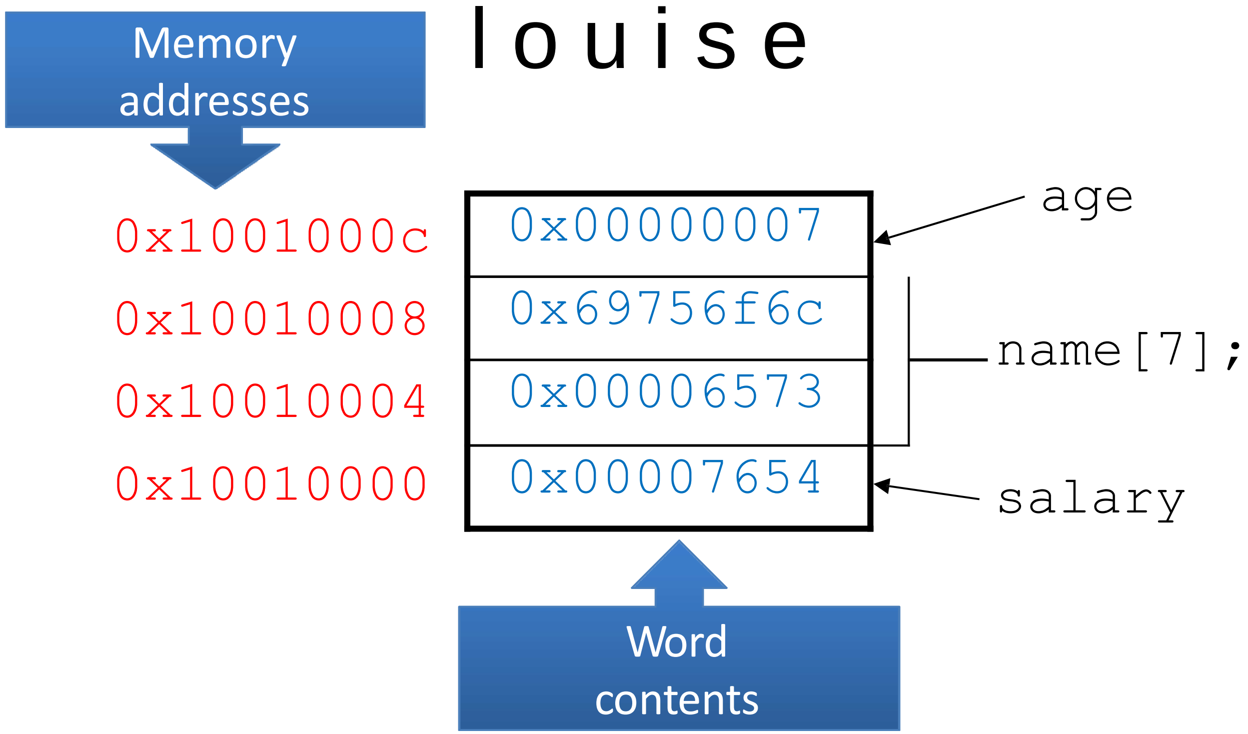
```
int age = 0x7; // hex
char name [7] = “louise”;
int salary = 0x7654; // hex
```

Our C program’s global declarations

- Will be laid out in memory as follows :

Remember:
4 bytes/word of memory
2 hex characters per byte

```
int age = 0x7;
char name [7] = “louise”;
int salary = 0x7654;
```



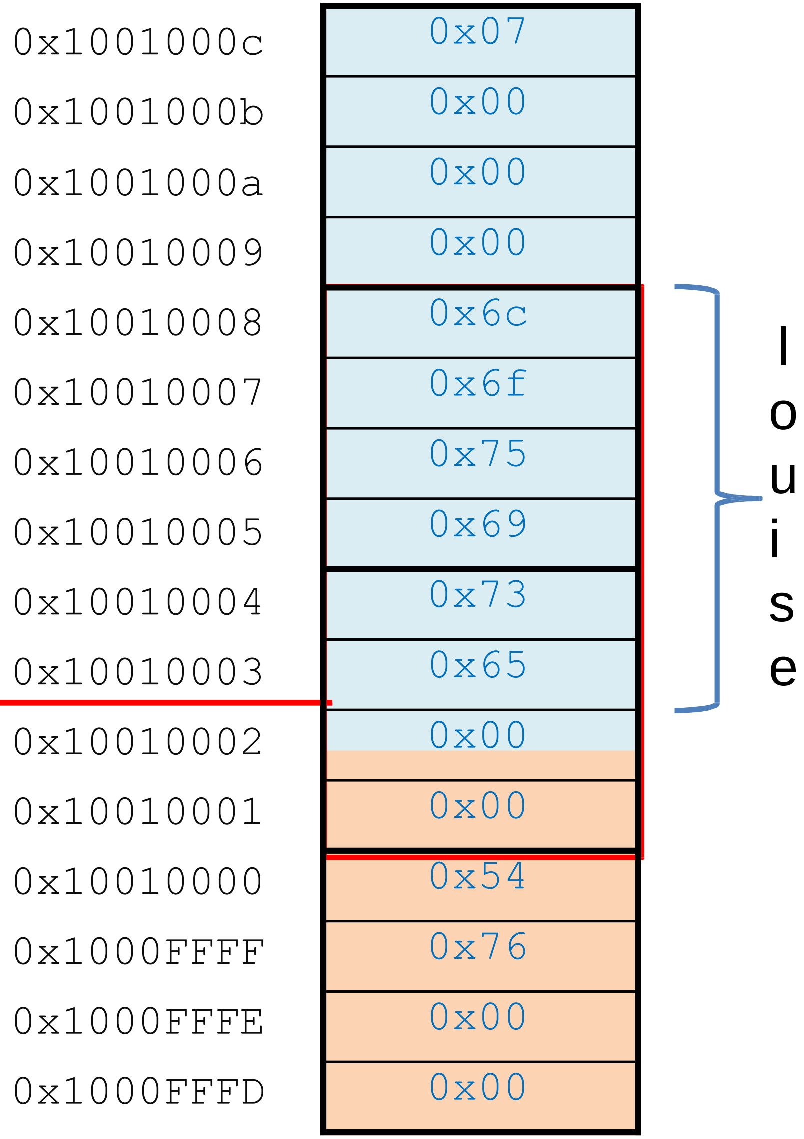
‘l’	‘o’	‘u’	‘i’	‘s’	‘e’	‘\0’
6c	6f	75	69	73	65	0

ASCII codes

With byte addressing

Why `name[7]` takes up 8 bytes memory????

Hints:
The word-size of this machine is 4 bytes



LOCAL VARIABLES & STACK

How is space allocated for Local Variables?

- Local variables only exist at runtime when the function, procedure or method containing them is being called.
- We need some mechanism that allows us to
  - allocate space when a function, procedure or method is called, and
  - deallocate space (so it can be reused) when we return from a procedure

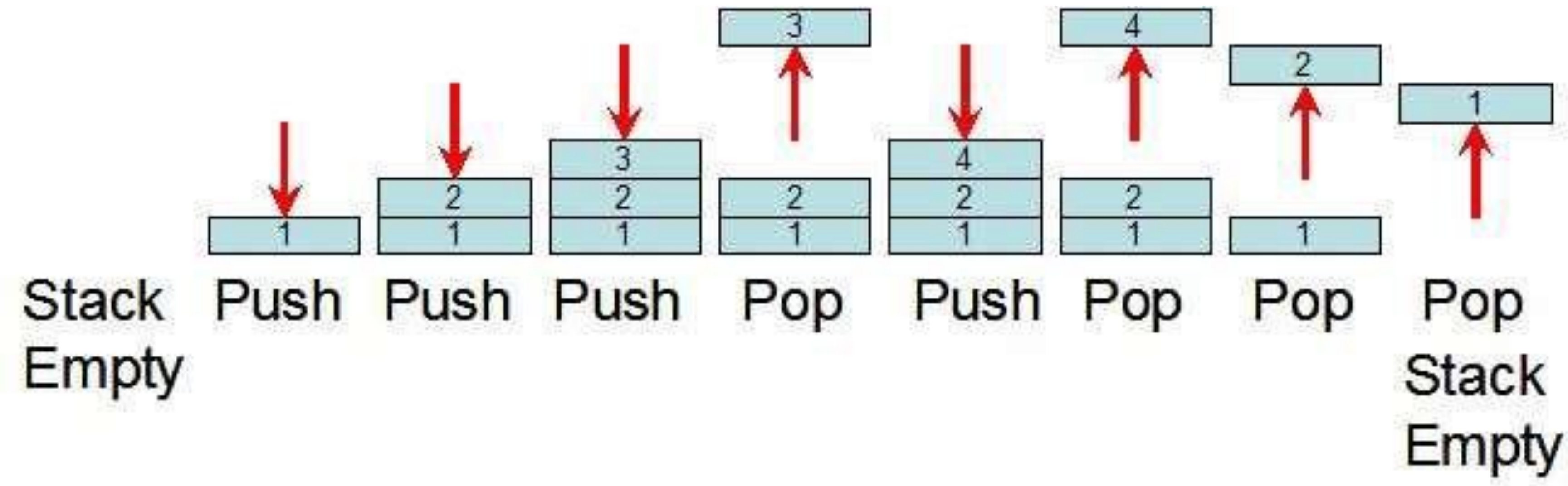
Stacks

- What is a Stack?
  - A stack is a data structure of ordered items such that items can be inserted and removed only at one end.

Stacks (cont)

- What can we do with a stack?
  - `push` - place an item on the stack
  - `pop` - Look at the item on top of the stack and remove it
  - `isEmpty` - Reports whether the stack is empty or not

Popping and pushing



Quiz

- We can use a stack to reverse the letters of a string.
  – E.g. “Hello World!” -> “!dlroW olleH”
  – How?

Stacks (cont)

- Problem:
  - What happens if we try to pop an item off the stack when the stack is empty?
    - This is called a stack underflow.
    - The pop method needs some way of telling us that this has happened.

A SIMPLE INT STACK IN ‘C’

pushing and popping (1)

push(x);	
push(y);	
x = pop ();	
y = pop ();	

sp	100
x	5
y	6

stack

99	??
98	??
97	??
96	??
…	??
0	??

```
int stack[100];
int sp = 100; //Stack pointer: NOT a
              // pointer data type

void push (int value)
{
   sp = sp-1;
   if (sp < 0) return; //overflow
   stack[sp] = value;
}
```

pushing and popping (2)

`push(x);`	*
`push(y);`	
`x = pop ();`	
`y = pop ();`	

sp	99
x	5
y	6

stack

99	5
98	??
97	??
96	??
…	??
0	??

```
void push (int value)
{
  sp = sp-1;
  if (sp < 0) return; //overflow
  stack[sp] = value;
}
```

sp : 100 -> 99
stack[99]=5

pushing and popping (3)

push(x);	
push(y);	*
x = pop ();	
y = pop ();	

sp	99
x	5
y	6

stack

99	5
98	6
97	??
96	??
…	??
0	??

sp : 99 -> 98

stack[98]=6

```
void push (int value)
{
  sp = sp-1;
  if (sp < 0) return; //overflow
  stack[sp] = value;
}
```

pushing and popping (4)

push(x);	
push(y);	
x = pop ();	*
y = pop ();	

sp	98
x	5
y	6

stack

99	5
98	6
97	??
96	??
…	??
0	??

```
int pop()
{
        if (sp >= 100) return -1; //stack underflow!
        int value = stack[sp];
        sp = sp + 1;
        return value;
}
```

value = stack [98] = 6

sp: 98 -> 99

pushing and popping (5)

push(x);	
push(y);	
x = pop ();	
y = pop ();	*

sp	99
x	5
y	6

stack

99	5
98	??
97	??
96	??
…	??
0	??

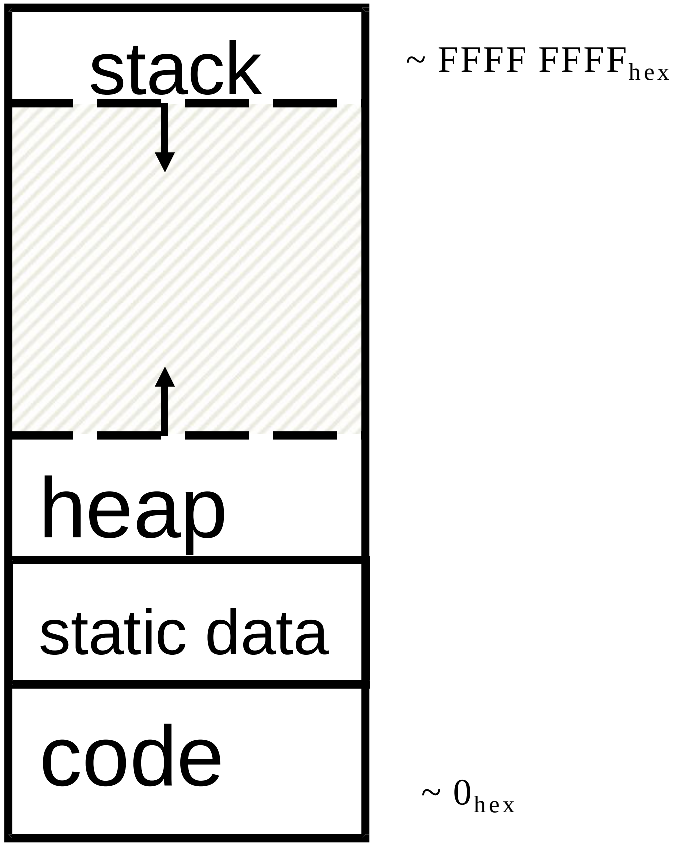
```
int pop()
{
        if (sp >= 100) return -1; //stack underflow!
        int value = stack[sp];
        sp = sp + 1;
        return value;
}
```

value = stack [99] = 5

sp: 99 -> 100

The run-time Stack

- Assume the stack begins at the high-end of the data portion of memory.
- As items are added, the stack pointer is decremented.
  - "grow downwards" towards the program section of memory.



How Local Variables are pushed onto the stack?


```
int age = 0x7; // hex
char name [6] = “louise”;
int salary = 0x7654; // hex

void doIncrement()
{
        int increment = 0x10; // hex
        salary = salary + increment;
}
void main()
{
        char reverse [7];
        doIncrement();
        printf(“salary plus increment = %d\n”,salary);
}
```

Making room for `reverse[7]`

- Assume the stack pointer is set to 1000
- There are 7 bytes in `reverse`, but we will round this up to a word boundary
  - hence we must reserve 8 bytes on the stack …
  - `sp = 992` after allocating space for `reverse[7]`

1000	??
999	??
998	??
997	??
996	??
995	??
994	??
993	??
992	??
991	??

Calling doincrement

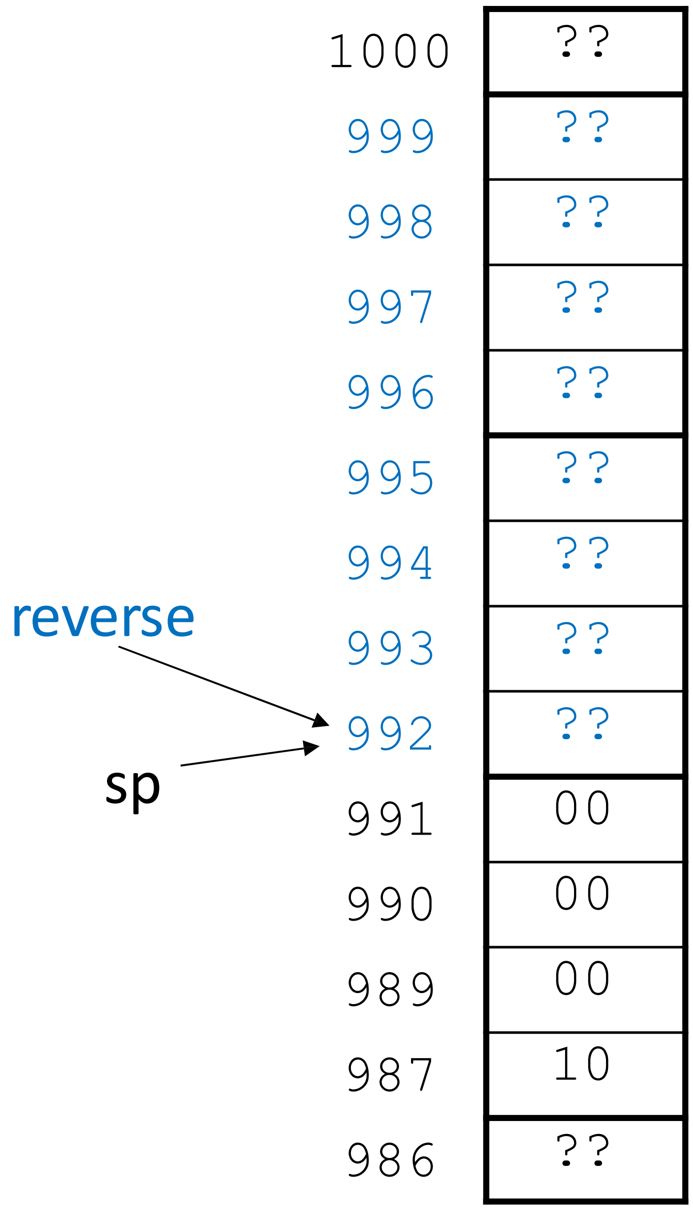
```
int age = 0x7; // hex
char name [6] = "louise";
int salary = 0x7654; // hex

void doIncrement()
{
        int increment = 0x10; // hex
        salary = salary + increment;
}

void main()
{
        char reverse [7];
        doIncrement();
        printf("salary plus increment = %d\n",salary);
}
```

Calling doincrement (2)

```
int age = 0x7; // hex
char name [6] = “louise”;
int salary = 0x7654; // hex

void doIncrement()
{
        int increment = 0x10; // hex
        salary = salary + increment;
}

void main()
{
        char reverse [7];
        doIncrement();
        printf(“salary plus increment = %d\n”,salary);
}
```

Calling doIncrement

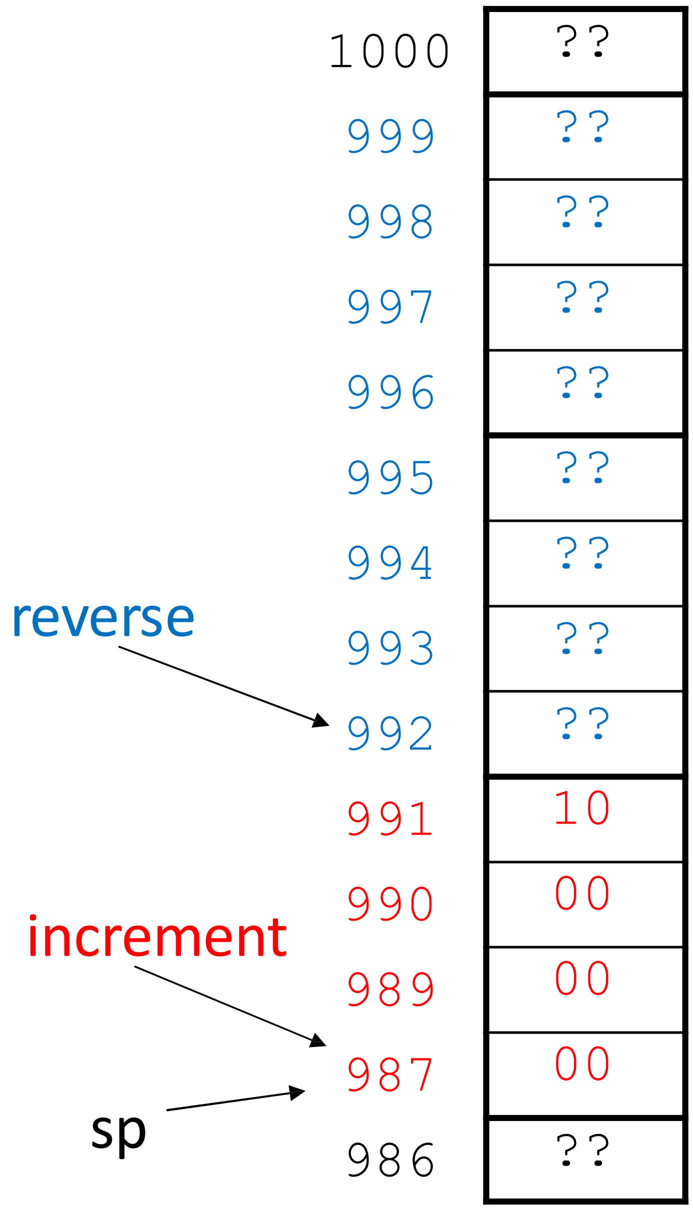
- `doIncrement` contains a local variable,
  `int increment = 0x10`
- We need to allocate space (4 bytes) for this on the stack.



Calling doIncrement

- `doIncrement` contains a local variable,
  `int increment = 0x10`
  - We need to allocate space (4 bytes) for this on the stack.

If `doIncrement` had any parameters, we would also have to allocate space for them on the stack.



Managing procedure calls

- Whenever a subroutine (procedure, function) is called, some storage must be set aside for
  - local variables
  - parameters (or arguments)
  - management information i.e. the return address

the call

```
void main()
{
        char reverse [7];
        foo(20);
        printf(“salary plus increment = %d\n”,salary);

}
```

At the end of the call, control returns to here

Stack frame

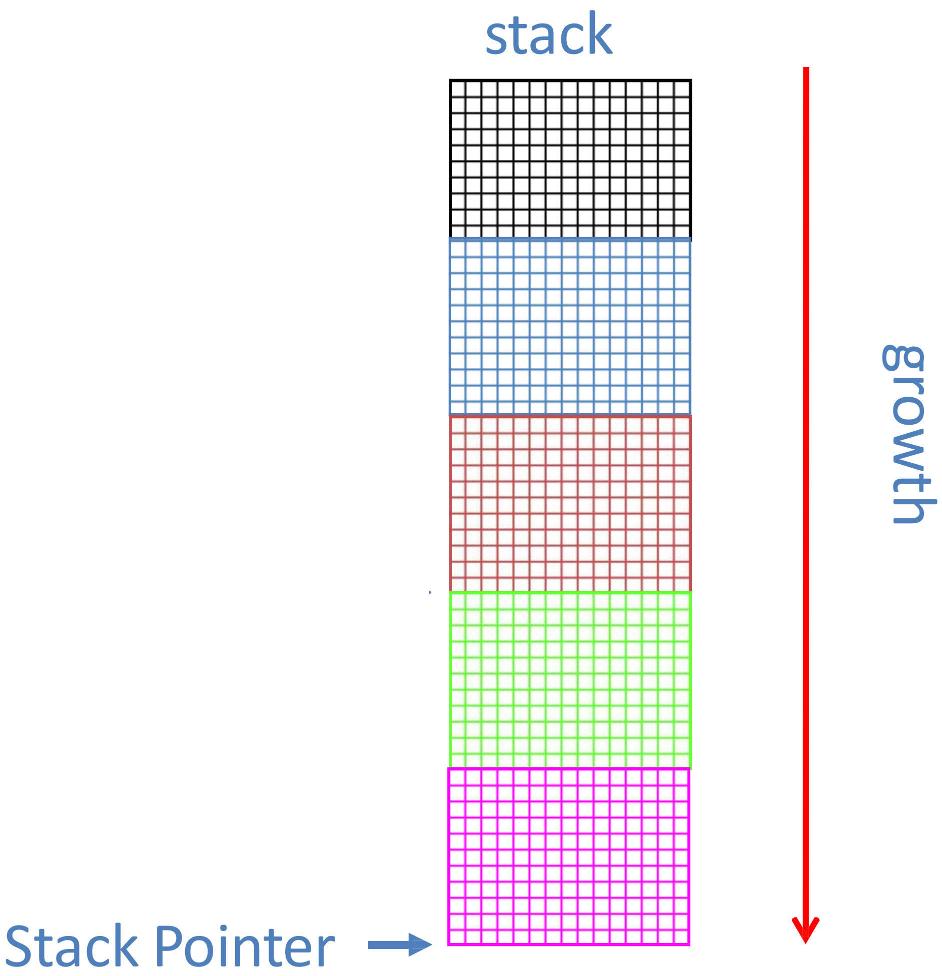
- A stack frame is an area that stores:

Copies of values of parameters
All local variables
Return result (if any)
Return address

- When a function is called, a new stack frame is created and pushed on the stack.
- And every time we exit a function, the stack frame is popped off the stack.

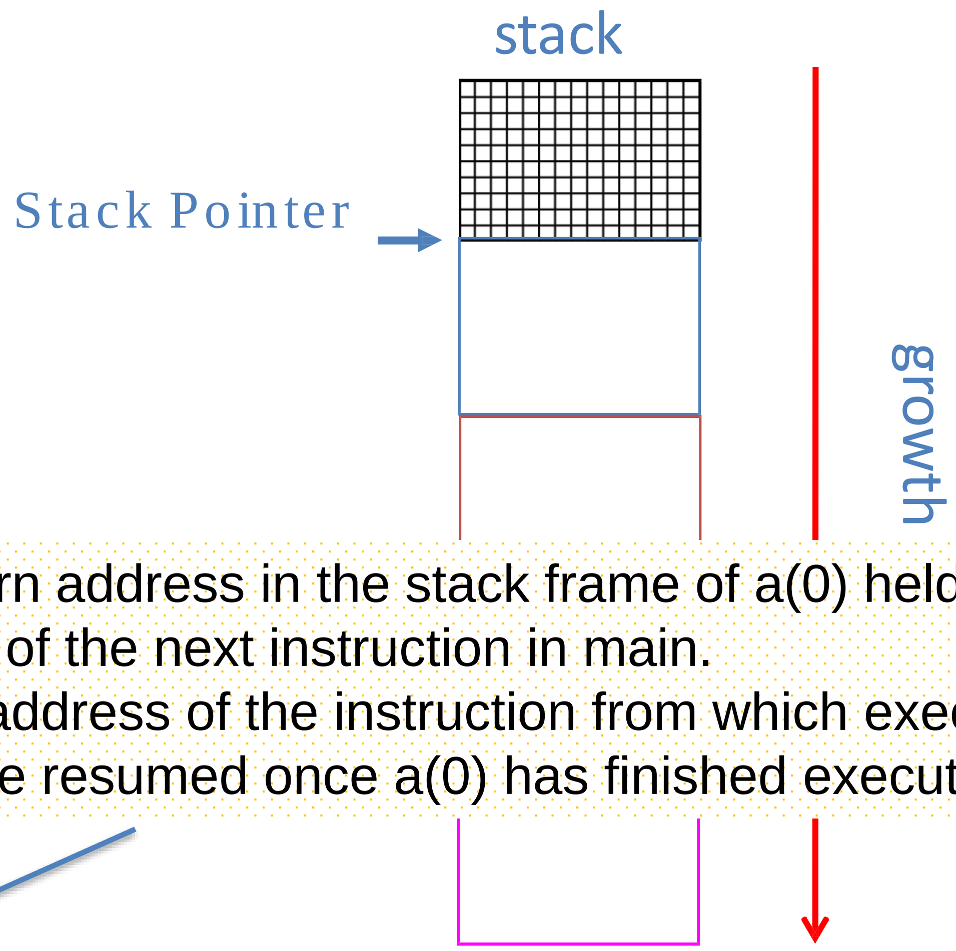
Creating Stack Frames

```
void d (int p)
{
}

void c (int o)
{ d(o);
}

void b (int n)
{ c(n);
}
void a (int m)
{ int a = m+1; b(a);
}
main ()
{
        a(0);
        printf("hello");
}
```



Creating Stack Frames

```
void d (int p)
{
}

void c (int o)
{ d(o);
}

void b (int n)
{ c(n);
}
void a (int m)
{ int a = m+1; b(a
}
main ()
{
        a(0);
        printf("hello");
}
```



The return address in the stack frame of a(0) held the Address of the next instruction in main.
It is the address of the instruction from which execution should be resumed once a(0) has finished executing.

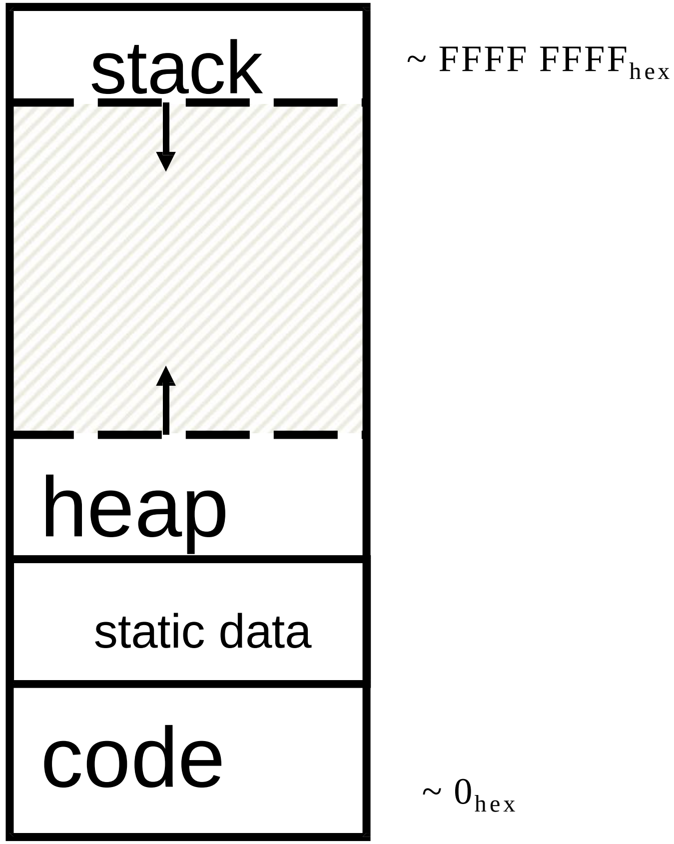
Unit 9:
Dynamic storage structures and memory management

Aims of this unit

- At the end of this unit you will understand :
  - The heap :
    - the area of memory used to dynamically allocate (e.g. malloc) and free memory cells, as used in dynamic data structures such as linked lists
  - How the heap space can be managed, both by
    - Automatic garbage collection (used by the language system)
    - System calls (used directly by the programmer)

Memory Management Revisit

- stack
  - Variable size, variable content (local variables etc.)
  - Used for managing function calls and returns
- heap
  - Dynamically allocated objects and data structures
  - data lives until deallocated
- static data
  - Fixed size, fixed content, allocated at compile time
- code
  - loaded when program starts, does not change



The Heap (A Typical Architecture)

- Contains a linked list of used and free cells (blocks)
- New cells are created as needed, e.g., using new (in java) or malloc (in C)
- Redundant (no in-used) cells should be released
  - Question: How can redundant cells be released?

Memory blocks can be different in sizes

- In a hybrid Java/'C' style :

```
struct IntCell
{
    int data;       //4 bytes
    struct IntCell* next;
    //32 bit addressing - 4 bytes
};
```

```
IntCell ic = new IntCell();
//new creates an instance of
IntCell
```

Memory blocks can be different in sizes (2)


```
struct PersonCell
{
    char name [12];             // 12 bytes
    int age;                    // 4 bytes
    struct IntCell* next;       // 4 bytes
};
```

- A PersonCell instance has 12 + 4 + 4 = 20 bytes.

```
PersonCell pc = new PersonCell();
```

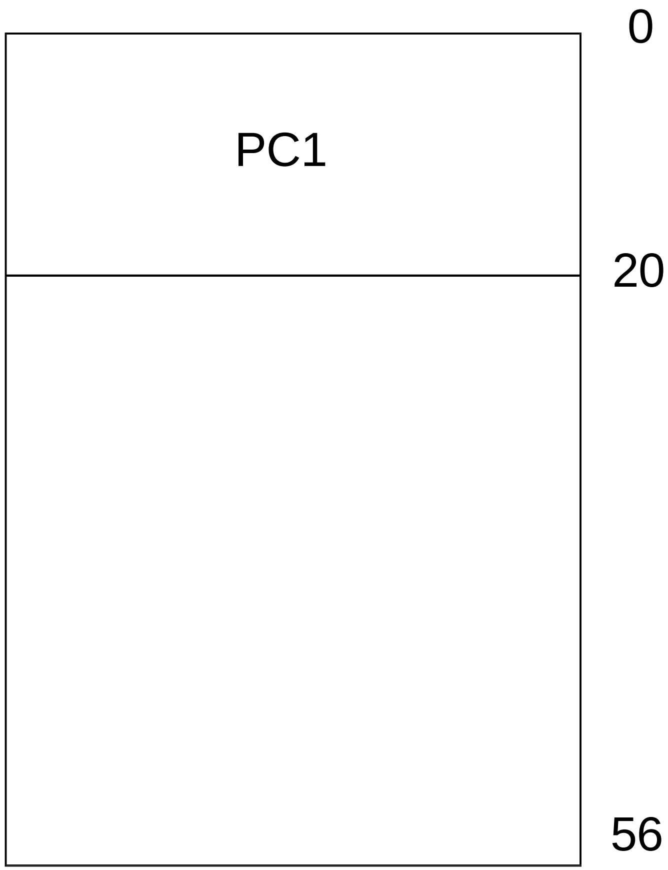
20 bytes allocated from the heap.

Memory Fragmentation -- Assumptions

- Assume a tiny heap with 56 bytes
  - just to keep our examples simple!
- We have six variables : PC1, PC2 and PC3 of type `PersonCell` (20 bytes each)
- and IC1, IC2 and IC3 of type `IntCell` (8 bytes each).

How fragmentation can arise (1)

```
PersonCell PC1 = new PersonCell();
```



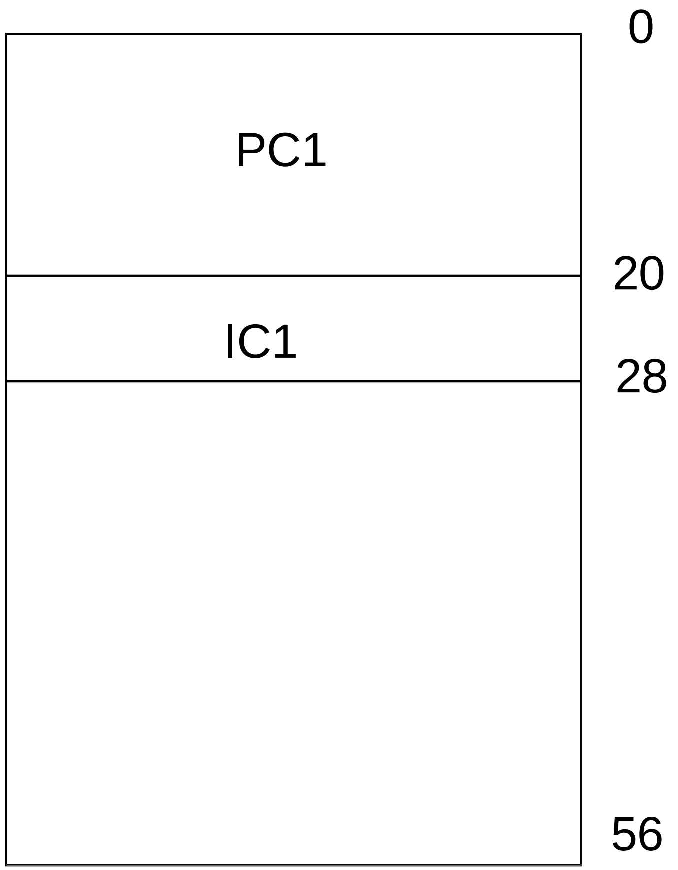
Free space : 56

Bytes required : 20

After allocation, free space left : 36

How fragmentation can arise (2)

```
IntCell IC1 = new IntCell();
```



Free space : 36

Bytes required : 8

After allocation,

free space left : 28

How fragmentation can arise (3)

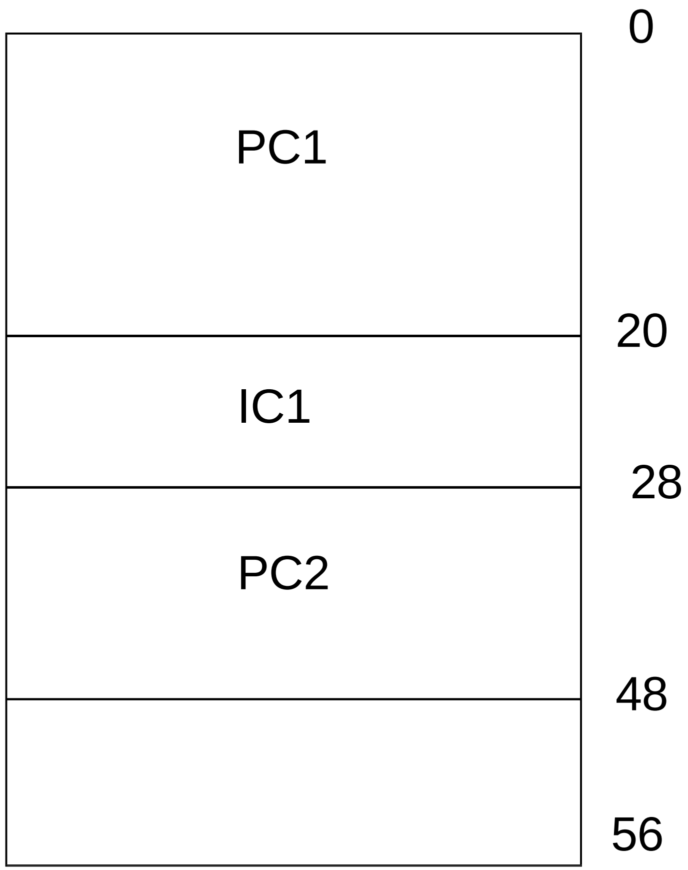

```
PersonCell PC2 = new PersonCell();
```

Free space : 28

Bytes required : 20

After allocation,

free space left : 8



How fragmentation can arise (4)

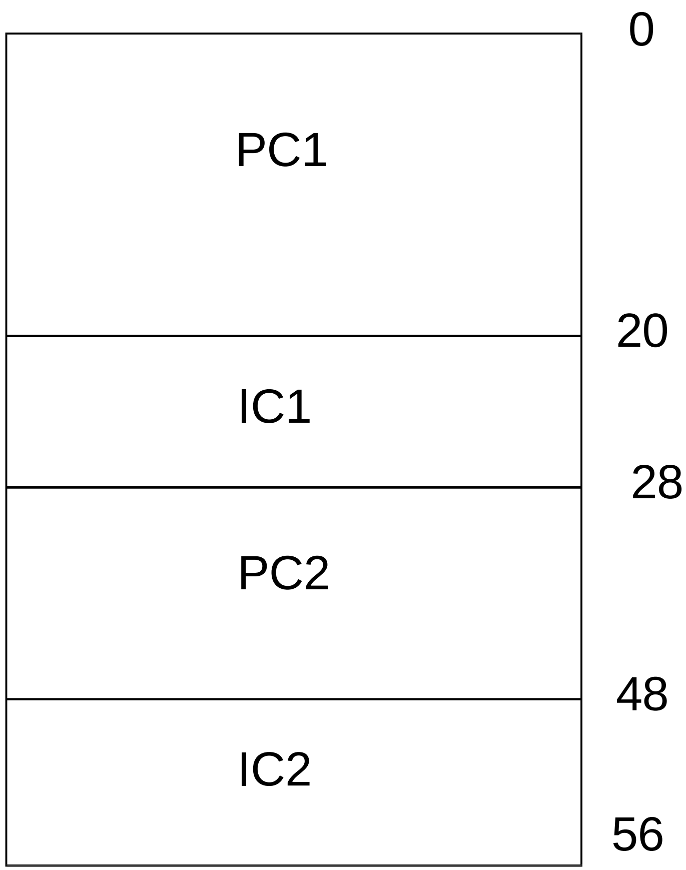
```
IntCell IC2 = new IntCell();
```

Free space : 8

Bytes required : 8

After allocation,

free space left : 0



How fragmentation can arise (5)

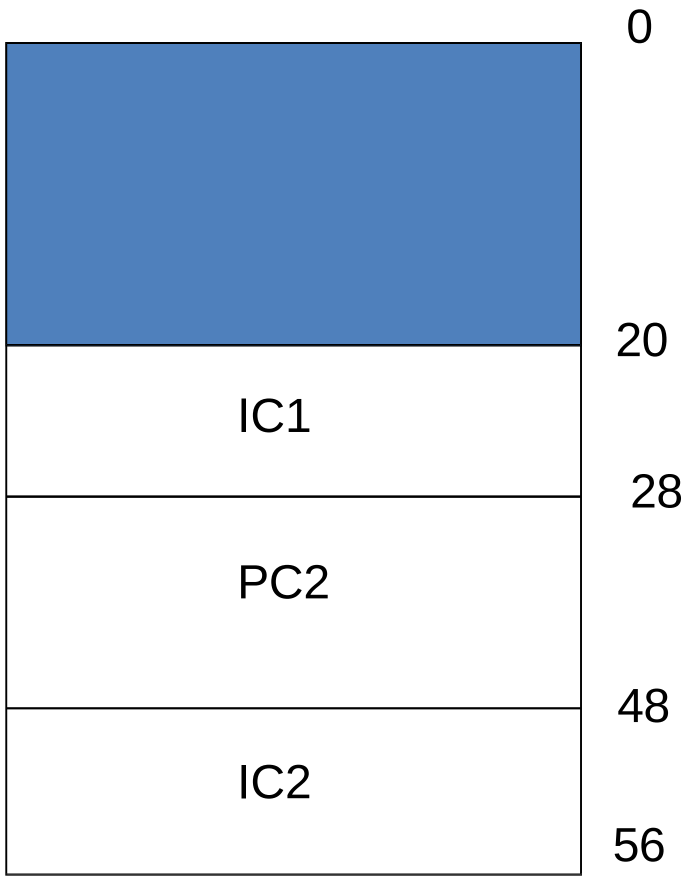
```
free(PC1); //not possible in Java!
```

Free space : 0

Bytes released : 20

After deallocation,

free space left : 20



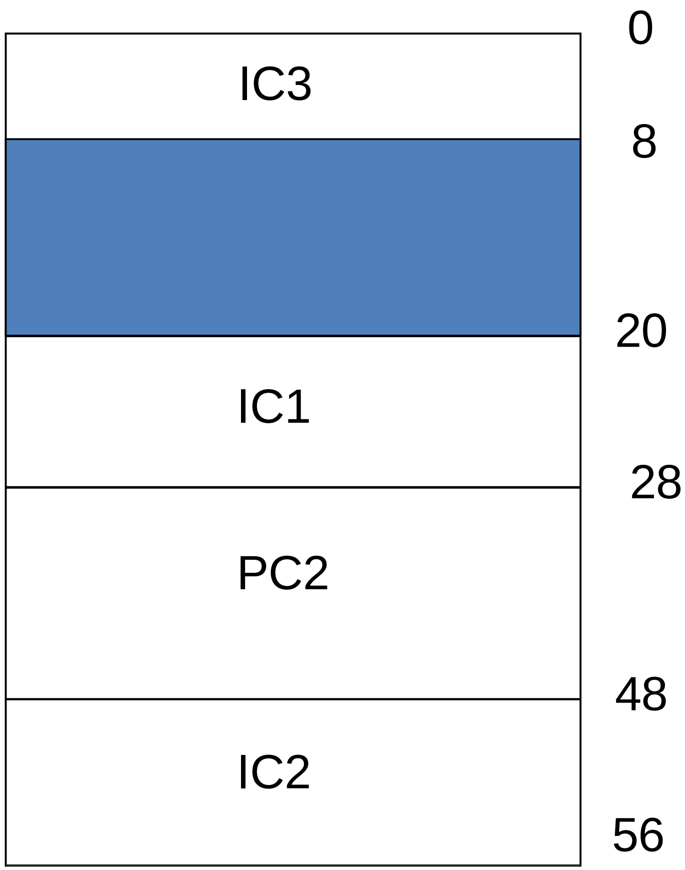
How fragmentation can arise (6)

```
IntCell IC3 = new IntCell();
```

Free space : 20

Bytes required : 8

After allocation, free space left : 12



How fragmentation can arise (7)

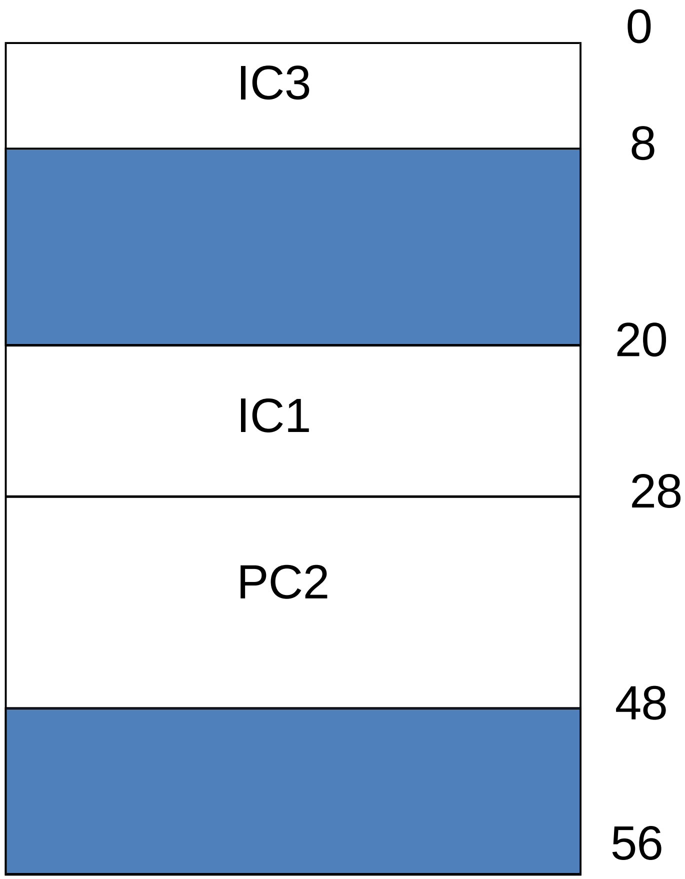

```
free(IC2);
```

Free space : 12

Bytes released : 8

After deallocation,

free space left : 20



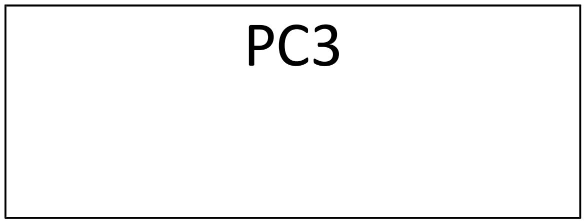
How fragmentation can arise (8)

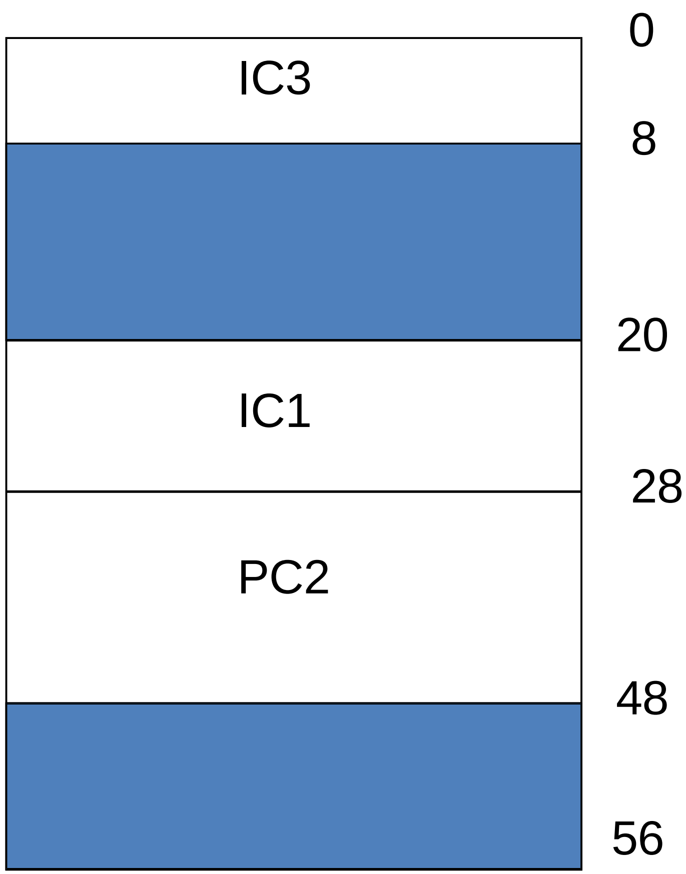
```
PersonCell PC3 = new PersonCell(); ???
```

Free space : 20

Bytes required : 20

But cannot allocate as the 20 bytes are not contiguous





‘C’ system calls : a reminder

- `malloc()`: Allocates raw, uninitialized memory from heap (malloc -> memory allocate)
- `sizeof` : `malloc` needs to know how many bytes to allocate;
  - given a C type as parameter, this returns how many bytes a variable of that type takes up.
- `free()` : deallocates memory

The Heap : Freeing Unwanted Cells

- Dangerous:
  - With `free` the programmer can free an in-use cell. (i.e. the cell is shared by two or more data structures)
- Resulting errors are
  - hard to detect
  - may be unnoticed
- These are some reasons why perhaps it is better for cells to be freed by the system rather than the programmer

GARBAGE COLLECTION

Garbage in Java

- Objects (data + methods) take up space
- Java automatically figures out which objects are not being used
- Cells (objects) in dynamic data structures in Java become “garbage” when no-one is pointing at them anymore.

Garbage in Java (2)

```
PersonCell pc = new PersonCell();
 //Object 1
PersonCell pc2 = pc;
//pc2 refers to object 1 too
pc.age = 20;
store_to_a_file(pc)

//now finished with the object 1

pc = new PersonCell(); //object 2
pc2 = new PersonCell(); //object 3

// pc and pc2 now all point to a new
object, so object 1 becomes garbage
```

The Heap : Garbage Collection

- Garbage Collector
  - Finds redundant cells automatically, so programmer need not worry
  - Called as needed, or in background
- Many different approaches
  - But there is a performance and space overhead
  - Used by (e.g.) Java, Python, Scala…

The Need for Garbage Collection

- At a given moment, the heap contains:
  - cells which are live (in-use)
  - cells which are redundant (formerly in use) :
  - cells which are known to be free
- If creation of an object fails:
  - look for redundant cells
  - designate them to be free
  - try to create the object again

Mark & Sweep Garbage Collection

- Marking
  - Mark objects found as in-use (live);
    - E.g. Simply count the number of references to an object
  - finally, free all unmarked objects.
- Merging
  - If no free cell is large enough, due to fragmentation of the heap, merge adjacent free cells.
- Compaction
  - If still no large enough cell, move live cells to bottom of heap, leaving large free area at top.
  - Compaction is slow, due to need to revaluate pointers

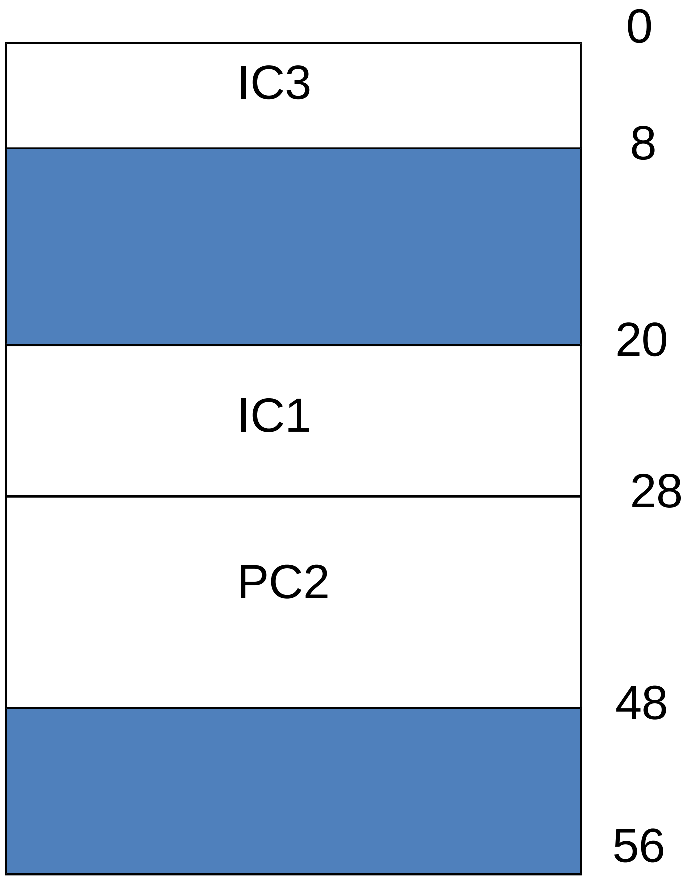
Compaction in practice (1)

Free space : 20

Bytes required : 20

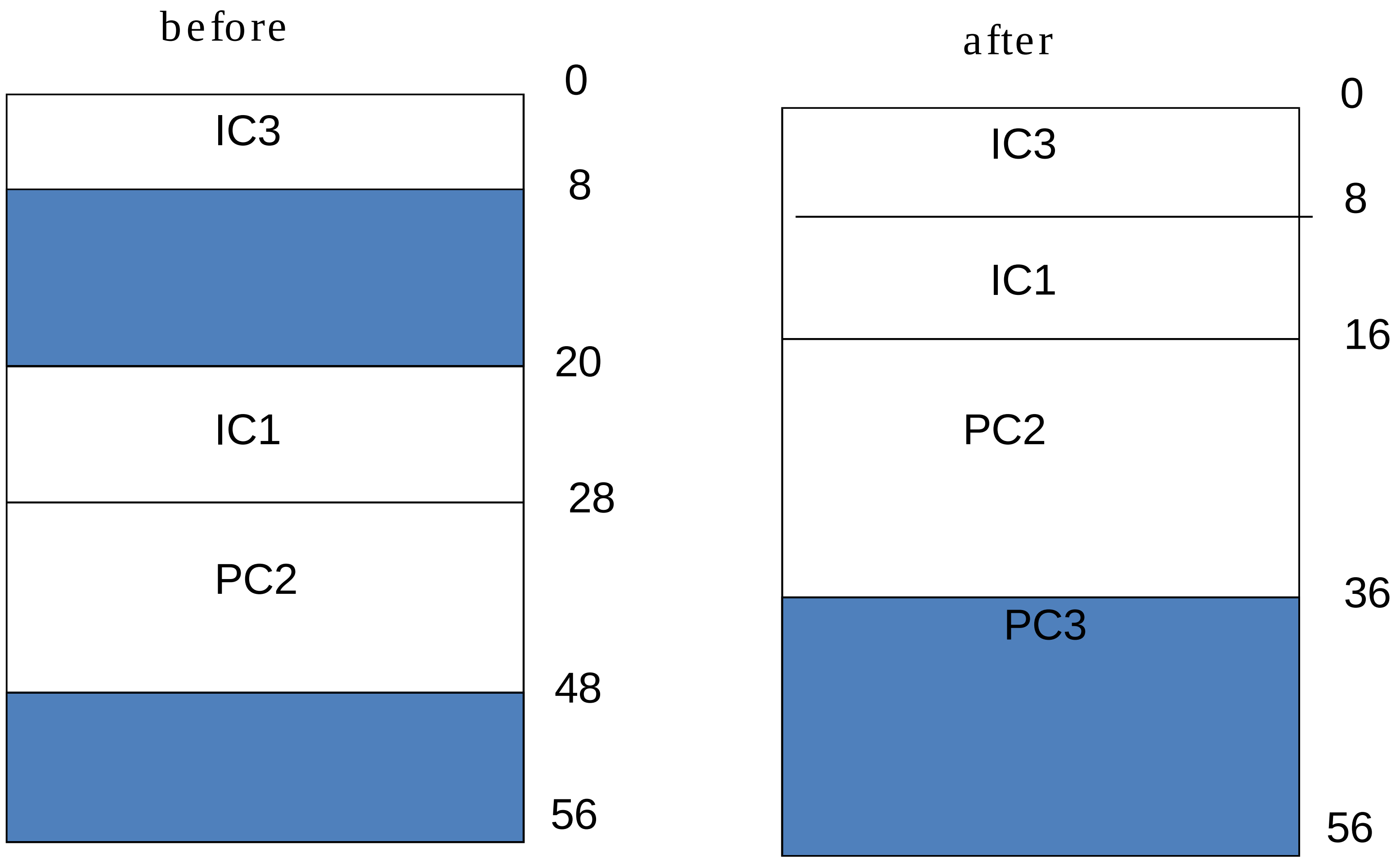
But cannot allocate as the 20 bytes are not contiguous

IC3 points to HEAP[0];
ICI points to HEAP[20];
PC2 points to HEAP[28];




```
PersonCell PC3 = new PersonCell(); ???
```

Compaction in practice (2)

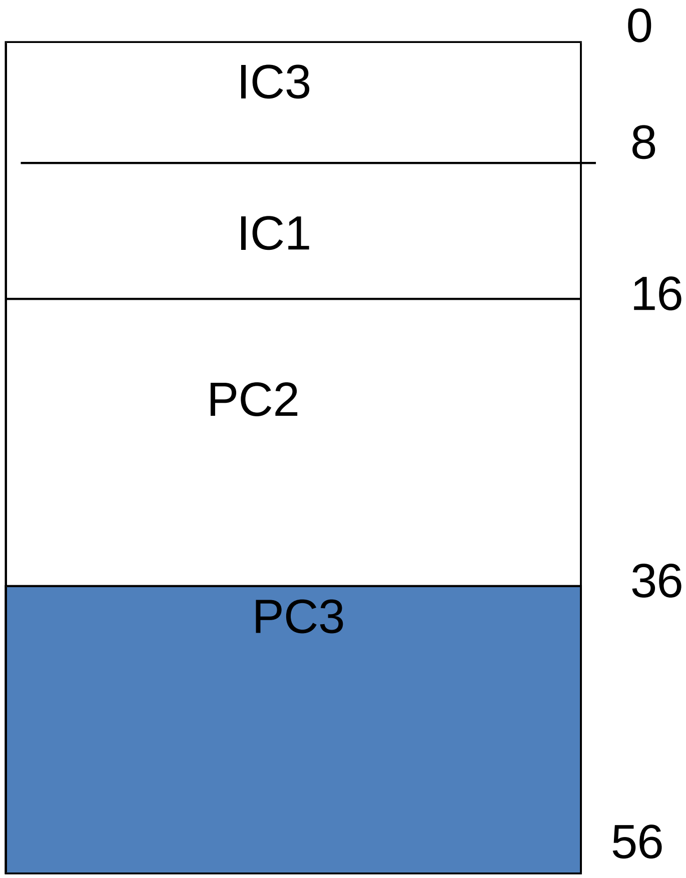



```
PersonCell PC3 = new PersonCell();
```

Compaction in practice (3)

Before compaction :
IC3 points to HEAP[0];
IC1 points to HEAP[20];
PC2 points to HEAP[28];

These have to be updated to
IC3 points to HEAP[0];
IC1 points to HEAP[8];
PC2 points to HEAP[16];




```
PersonCell PC3 = new PersonCell();
```

Garbage Collection in Java (cont)

- We can't explicitly call a "free" style method, but we can deliberately set a object to null.

```
PersonCell pc = new PersonCell(); //object 1
    …

pc = null;        // pc no longer points at
                  //the space allocated for object 1
```

- Also, if the object is a local variable on the stack, once we have popped off the containing stack frame, the variable no longer exists and thus it is no longer pointing at the space.

‘C’ Malloc/Free Implementation

- In ‘C’, how might the system calls `malloc` and `free` manage the heap?
- Each cell of memory is preceded by a header that has two fields:
  - size of the cell and
  - a pointer to the next cell
- All free cells are kept in a linked list

Simple Implementation

- `malloc()` searches the free list for a cell that is big enough. If none is found, more memory is requested from the operating system.
- `free()` checks if the cells adjacent to the freed cell are also free
  - If so, adjacent free cells are merged (coalesced) into a single, larger free cell
  - Otherwise, the freed cell is just added to the free list

Choosing a cell in malloc()

- How do we choose which free cell to use?
  - best-fit: choose the smallest cell that is big enough for the request
  - first-fit: choose the first cell we see that is big enough
  - next-fit: like first-fit but remember where we finished searching and resume searching from there
- SCC students will learn more about this in SCC.211 Operating Systems

The END