CASS Exercise session 4

Dynamic Memory

Pointers redux

pointers.c

```
#include <stdio.h>
int main()
{
   int i = 5;
   int* j = &i;
   int k = *j;
   printf("  Value of i: %i\n", i);
   printf("Address of i: %p\n", &i);
   printf("  Value of j: %p\n", j);
   printf("Address of j: %p\n", &j);
   printf("  Value of k: %i\n", k);
   printf("Address of k: %p\n", &k);
}
```

Console

```
$ - gcc pointers.c -o run-pointers
$ - ./run-pointers
  Value of i: 5
Address of i: 0x7f698878
  Value of j: 0x7f698878
Address of j: 0x7f698880
  Value of k: 5
Address of k: 0x7f69887c
$ -
```

Pointers redux

pointers.c

```
#include <stdio.h>
int main()
{
   int i = 5;
   int* j = &i;
   int k = *j;
   printf("  Value of i: %i\n", i);
   printf("Address of i: %p\n", &i);
   printf("  Value of j: %p\n", j);
   printf("Address of j: %p\n", &j);
   printf("  Value of k: %i\n", k);
   printf("Address of k: %p\n", &k);
}
```

Pointer declaration

*var – value stored under the address in var
&var – address of var

Memory

Address	Value
...	...
0x7ffe5f78	0x00..05
0x7ffe5f7c	0x00..05
0x7ffe5f80	0x7ffe5f78
...	...

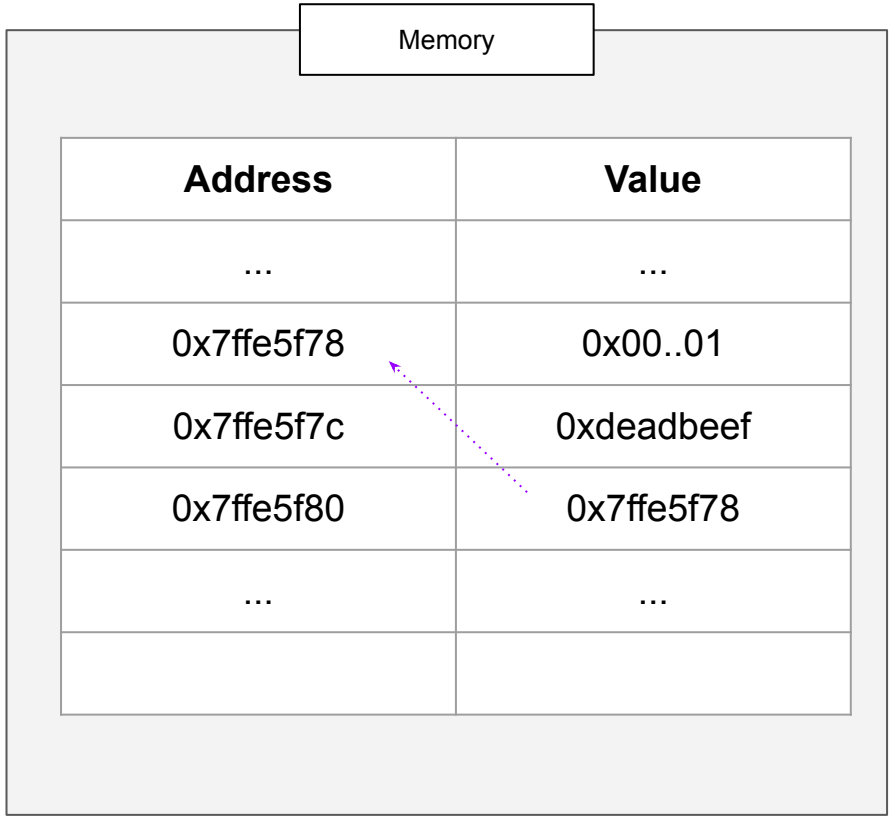
Order of variables chosen by the compiler

Pointer arithmetic

pointers.c

```
#include <stdio.h>
int main()
{
  int a = 1;
  char *c = “asdf”;
  int *p = &a;

  printf(“%p %p\n”, ++c, ++p);
}
```

Memory



Address	Value
...	...
0x7ffe5f78	0x00..01
0x7ffe5f7c	0xdeadbeef
0x7ffe5f80	0x7ffe5f78
...	...

Pointer arithmetic

What will happen after the increments?

pointers.c

```
#include <stdio.h>
int main()
{
  int a = 1;
  char *c = “asdf”;
  int *p = &a;

  printf(“%p %p\n”, ++c, ++p);
}
```

Memory

Address	Value
...	...
0x7ffe5f78	0x00..01
0x7ffe5f7c	0xdeadbeef
0x7ffe5f80	0x7ffe5f78
...	...

Pointer arithmetic

What will happen after the increments?

pointers.c

```
#include <stdio.h>
int main()
{
  int a = 1;
  char *c = “asdf”;
  int *p = &a;

  printf(“%p %p\n”, ++c, ++p);
}
```

Pointer c increases by 1

Memory

Address	Value
...	...
0x7ffe5f78	0x00..01
0x7ffe5f7c	0xdeadbef0
0x7ffe5f80	0x7ffe5f7c
...	...

Pointer arithmetic

What will happen after the increments?

pointers.c

```
#include <stdio.h>
int main()
{
  int a = 1;
  char *c = “asdf”;
  int *p = &a;

  printf(“%p %p\n”, ++c, ++p);
}
```

Pointer p increases by 4

Memory

Address	Value
...	...
0x7ffe5f78	0x00..01
0x7ffe5f7c	0xdeadbef0
0x7ffe5f80	0x7ffe5f7c
...	...

Pointer arithmetic

What will happen after the increments?

pointers.c

```
#include <stdio.h>
int main()
{
  int a = 1;
  char *c = “asdf”;
  int *p = &a;

  printf(“%p %p\n”, ++c, ++p);
}
```

Memory

Address	Value
...	...
0x7ffe5f78	0x00..01
0x7ffe5f7c	0xdeadbef0
0x7ffe5f80	0x7ffe5f7c
...	...

Each increment increases the pointer by sizeof(type)

Pointers and arrays

Arrays can be used as pointers

pointers.c

```
#include <stdio.h>
int main()
{
  int a[] = {1, 2, 3, 4};

  //a == &a[0] !!

  *a = 5; //OK
  a += 1; //BAD!

  int* b = a; //NOT &a! WHY?
  b++;   //OK
}
```

Memory

Address	Value
...	...
0x7ffe5f78	0x00..01
0x7ffe5f7c	0x00..02
0x7ffe5f80	0x00..03
0x7ffe5f84	0x00..04
...	...

Pointers and arrays

Arrays can be used as pointers

pointers.c

```
#include <stdio.h>
int main()
{
  int a[] = {1, 2, 3, 4};

  //a == &a[0] !!

  *a = 5; //OK
  a += 1; //BAD!

  int* b = a; //NOT &a! WHY?
  b++;   //OK
}
```

Memory

Address	Value
...	...
0x7ffe5f78	0x00..05
0x7ffe5f7c	0x00..02
0x7ffe5f80	0x00..03
0x7ffe5f84	0x00..04
...	...

Pointers and arrays

Arrays can be used as pointers

pointers.c

```
#include <stdio.h>
int main()
{
  int a[] = {1, 2, 3, 4};

  //a == &a[0] !!

  *a = 5; //OK
  a += 1; //BAD!

  int* b = a; //N
  b++;   //OK
}
```

Variable a of type int[4] is immutable!

Memory

Address	Value
...	...
0x7ffe5f78	0x00..05
0x7ffe5f7c	0x00..02
0x7ffe5f80	0x00..03
0x7ffe5f84	0x00..04
...	...

Pointers and arrays

Arrays can be used as pointers

pointers.c

```
#include <stdio.h>
int main()
{
  int a[] = {1, 2, 3, 4};

  //a == &a[0] !!

  *a = 5; //OK
  a += 1; //BAD!

  int* b = a; //NOT &a! WHY?
  b++;   //OK
}
```

Memory

Address	Value
...	...
0x7ffe5f78	0x00..05
0x7ffe5f7c	0x00..02
0x7ffe5f80	0x00..03
0x7ffe5f84	0x00..04
...	...

Pointers and arrays

Arrays can be used as pointers

pointers.c

```
#include <stdio.h>
int main()
{
  int a[] = {1, 2, 3, 4};

  //a == &a[0] !!

  *a = 5; //OK
  a += 1; //BAD!

  int* b = a; //NOT &a! WHY?
  b++;  //OK
}
```

Variable b of type int* is **not** immutable!

Memory

Address	Value
...	...
0x7ffe5f78	0x00..05
0x7ffe5f7c	0x00..02
0x7ffe5f80	0x00..03
0x7ffe5f84	0x00..04
...	...

Pointers and structs

pointers.c

```
struct person {
  int age;
  int grade;
};

int main()
{
  struct person s;
  struct person *p;

  p = &s;
  p->age = 13;   //Sets s.age to 13
  (*p).age = 13; //Same as above
}
```

- Use “->” instead of “.”
- Everything else remains the same

Problem: How to create a variable length array?

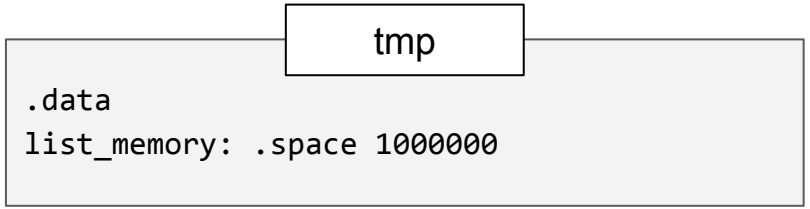
Lists

- ArrayList in Java, List in python, …
  - How can we implement these in a low level language?
  - Size not known at compile-time
  - Size is dynamic
    - Increases or decreases at runtime

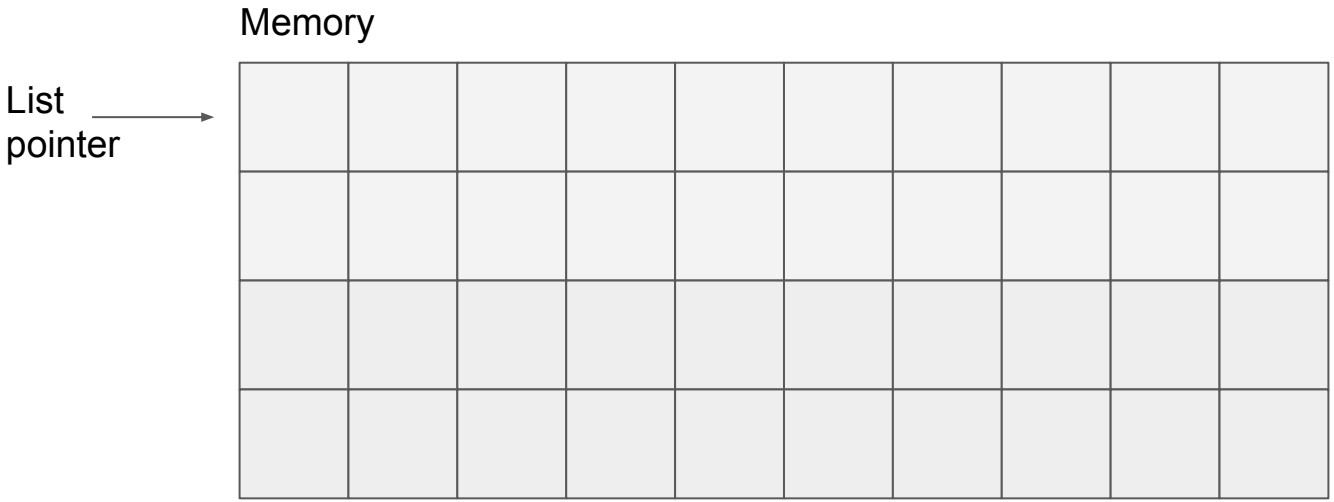
- Possible solution
  - Reserve a very large static array for every list
  - Problem: memory is wasted
  - Problem: what does “very large” mean? What if the list grows even bigger?

Lists

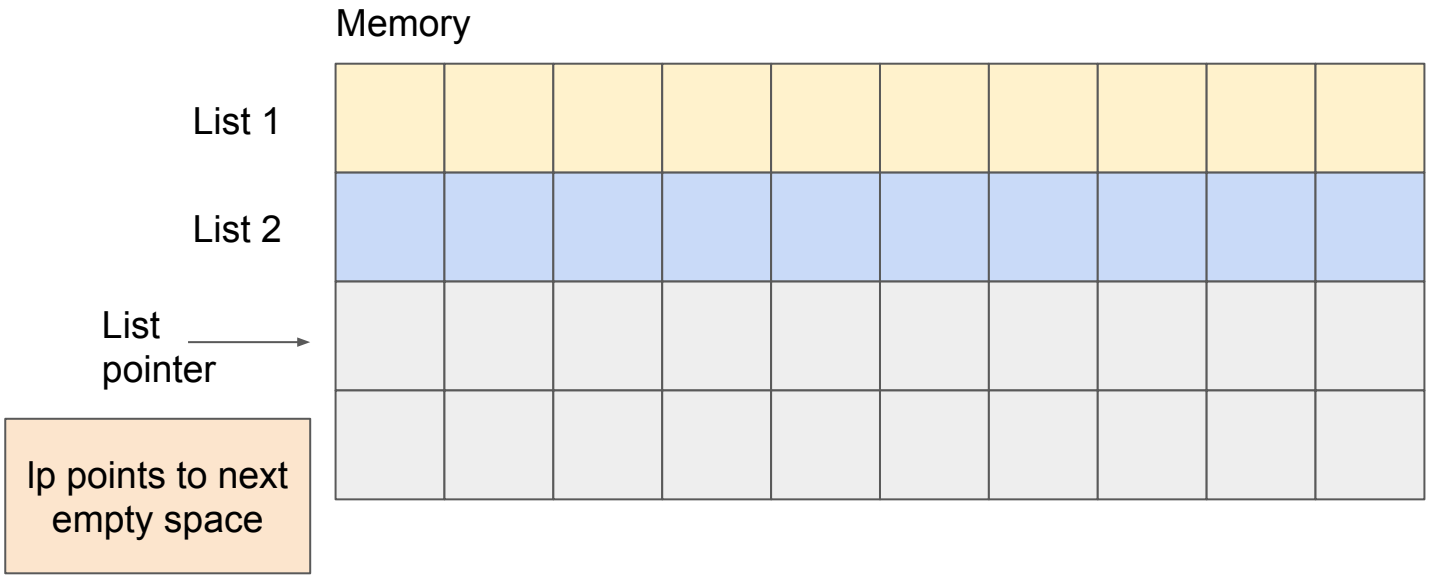
- Assume: program reserves big chunk of list memory
  - How can list implementations use this?



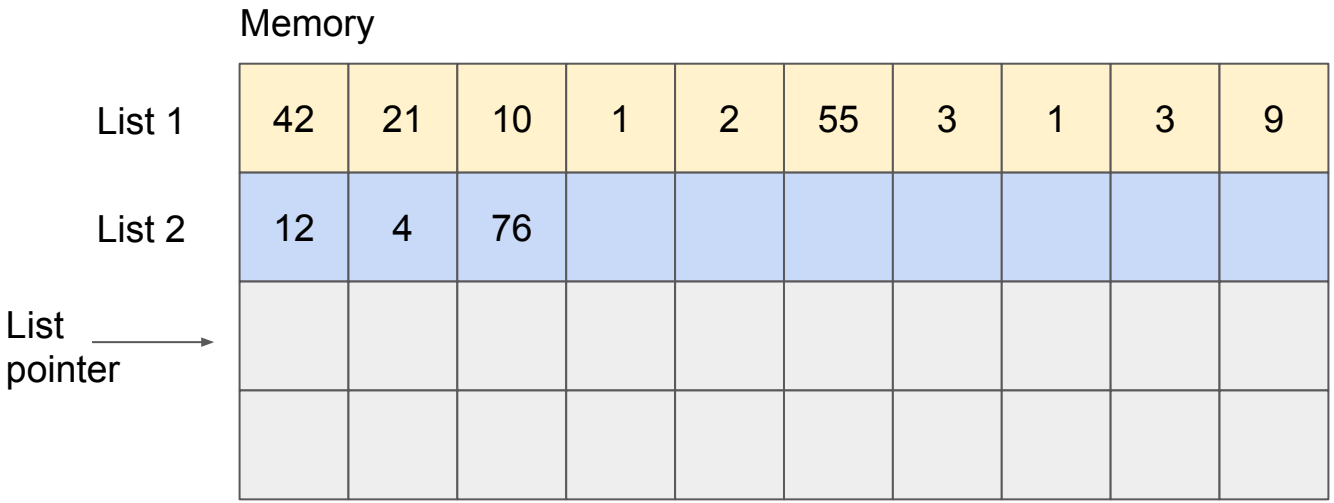
Idea: create a new list



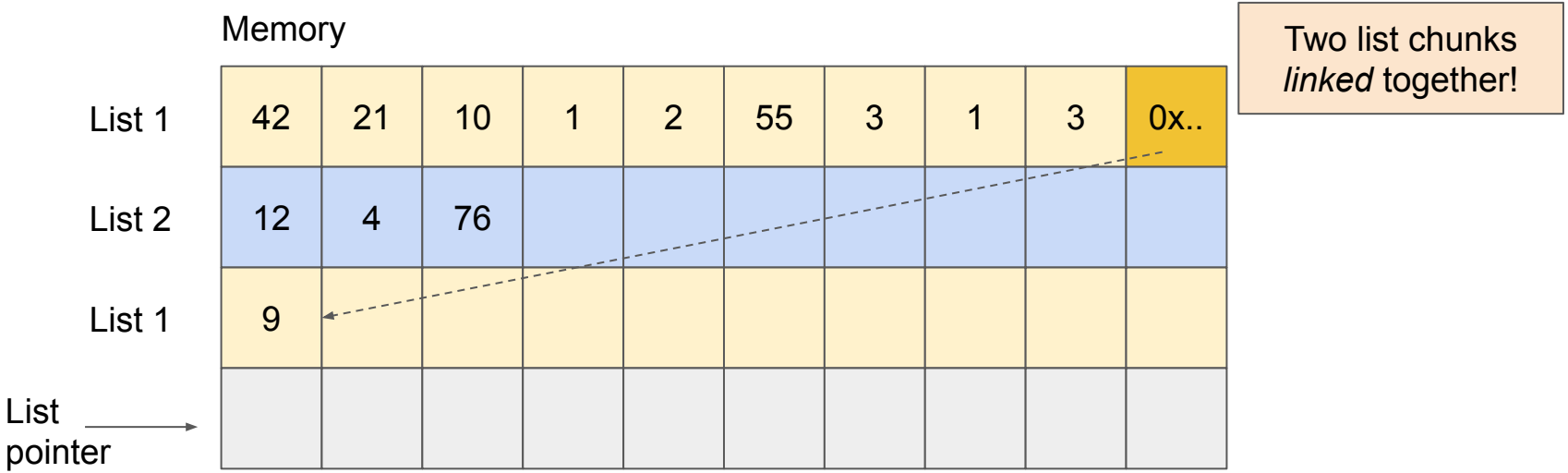
Idea: create a new list



Problem: list 1 is full



Solution: pointers!



Implementation

- Store list pointer in register
  - e.g. s9 (random choice!)
- Create list
  - Move list pointer
  - Return address of free space
- List full?
  - Create new one and link
  - Called a linked list

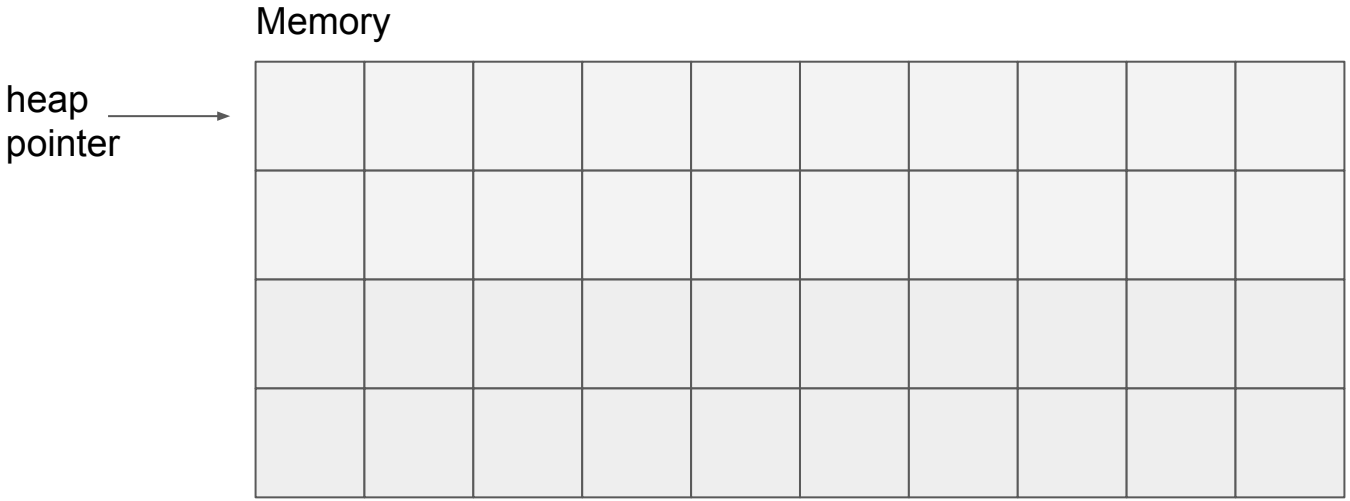
lists.asm

```
.data
list_memory: .space 1000000
.globl main

.text
#s9 keeps track of the list allocation
create_list:
    mv    a0, s9
    addi  s9, s9, 40
    ret

main:
    la s9, list_memory
    jal create_list
    mv t0, a0 #t0 now has pointer to list
    #expand
    jal create_list
    sw a0, 36(t0)
```

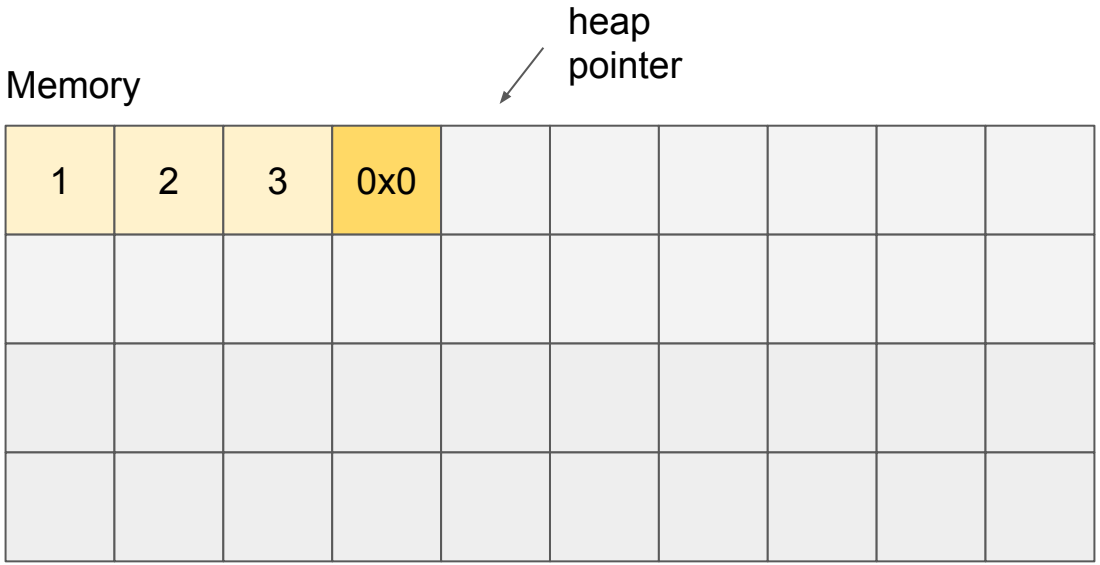
Problem: solution specific to lists

- All that space just for lists
- Can we generalize this for any growing, dynamic data structure?

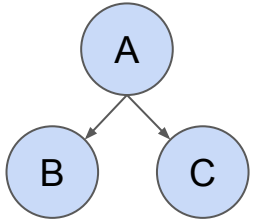
Solution: easy!



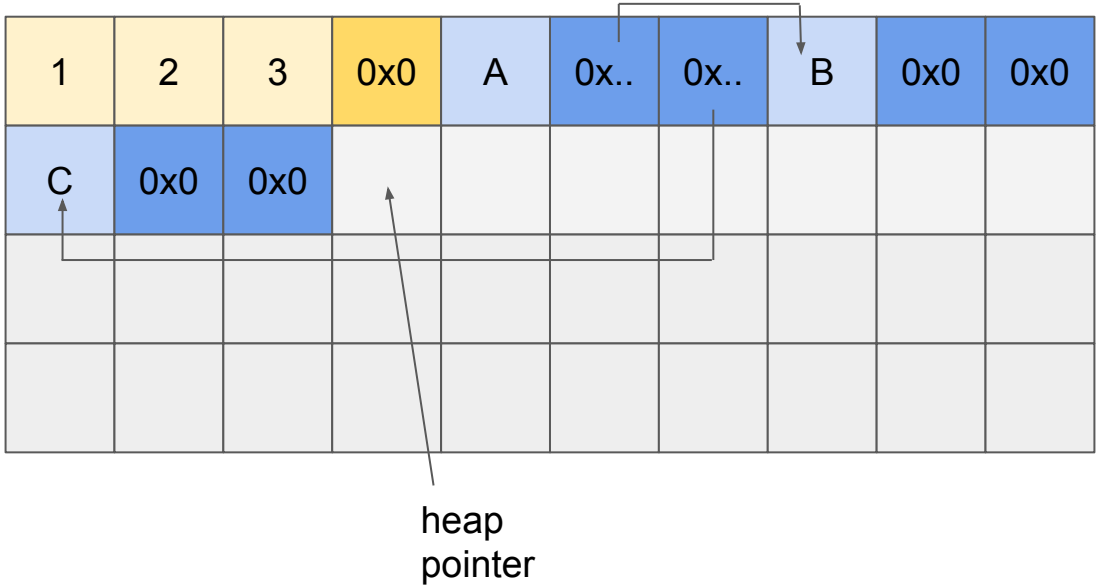
1. Allocate a list chunk



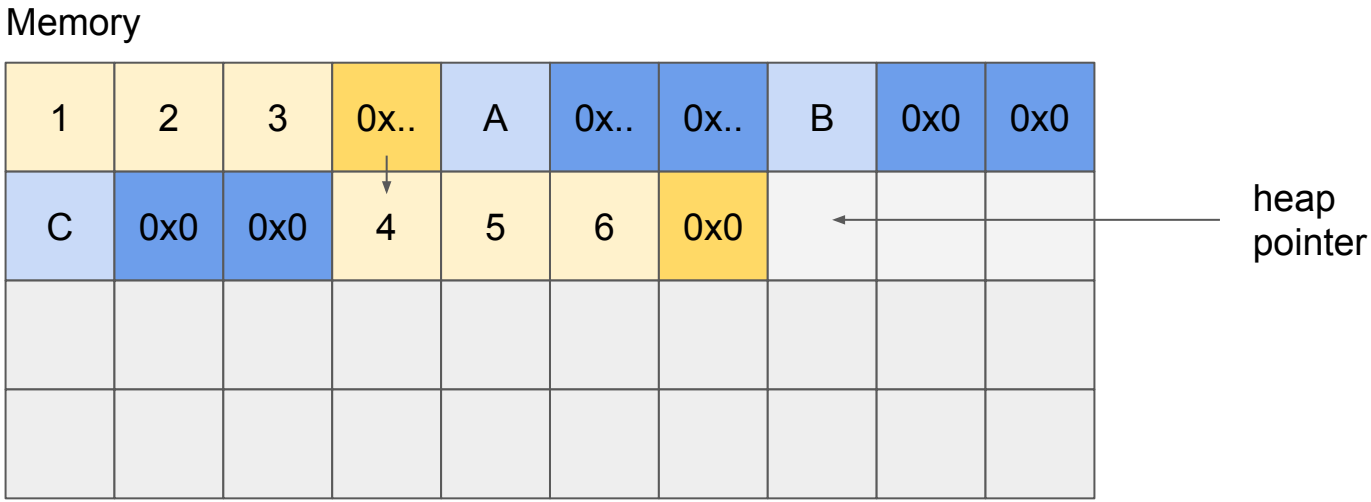
2. Allocate some binary tree chunks



Memory



3. Another list chunk



Implementation

- Generalize by adding parameter to allocation function
  - Can now reserve any amount of bytes
- gp register is used by convention
  - Caller-save or callee-save??
  - Nobody saves gp!

heap.asm

```
.data
heap: .space 1000000

.globl main
.text

#gp keeps track of the heap allocation
allocate_space:
    mv    t0, a0
    mv    a0, gp
    add   gp, gp, t0
    ret

create_list:
    li     a0, 40
    addi   sp, sp, -4
    sw     ra, (sp)
    jal    allocate_space
    lw     ra, (sp)
    addi   sp, sp, 4
    ret

main:
    la     gp, heap  #initialize gp!
    jal    create_list
```

Discussion

- allocate_space: Super simple allocator
  - Problem: how to free memory?
    - Long running programs will run out of memory
    - e.g. web server
  - Solution can be complex!

- How to do this in C?
  - C provides library functions
    - void *malloc(size_t bytes);
    - void free(void *ptr);
  - Complex allocators
    - Allows freeing previously allocated memory!