Introduction to Dynamic Memory Management

FACULTY OF
ENGINEERING

Dr. John Stavrakakis

COMP2017/COMP9017

THE UNIVERSITY OF
SYDNEY

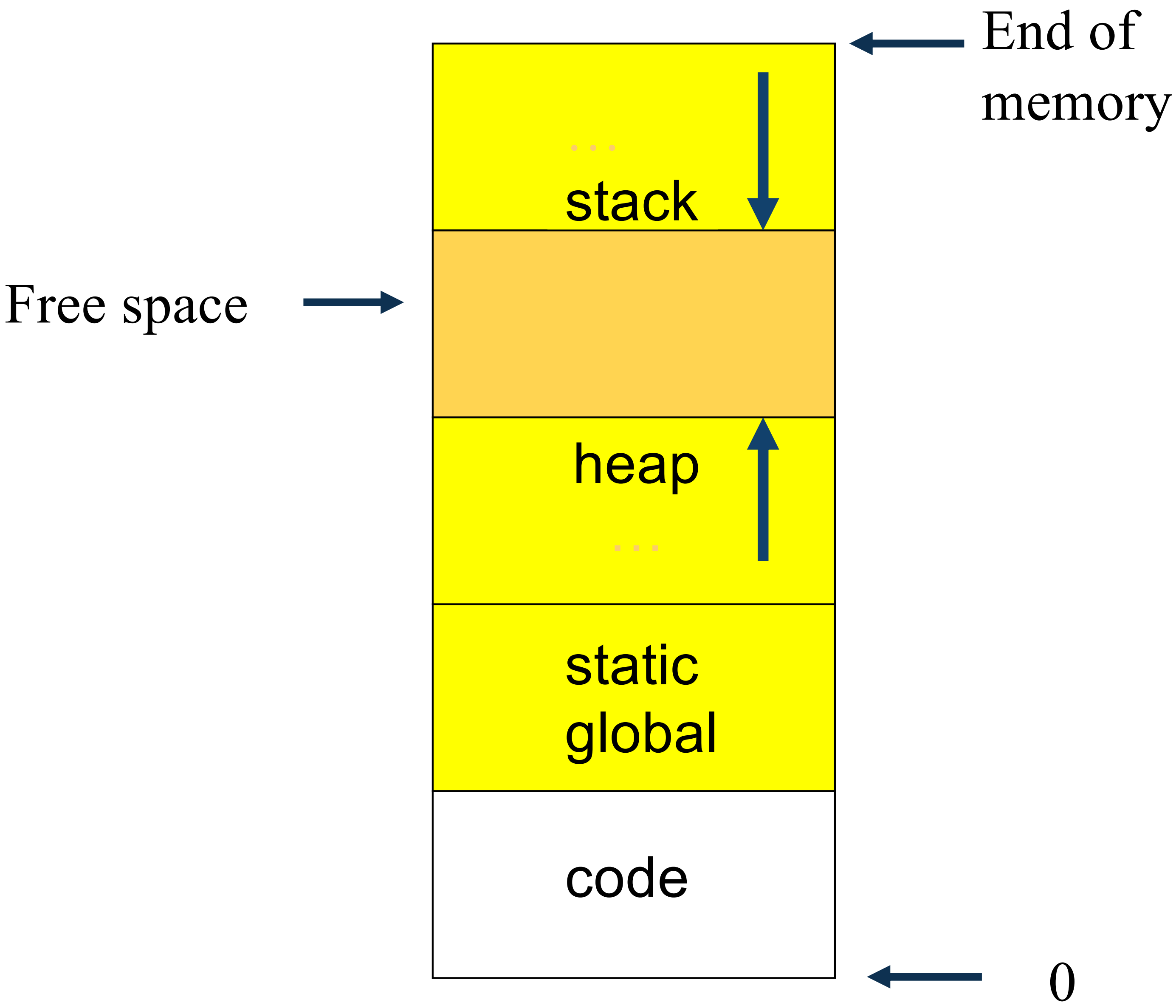
Memory

- Memory is a long array of 8 bit pieces called ***bytes***
- This array is indexed from 0 to the number of bytes in the memory
- Each index is a memory ***address***

Memory Areas

- Stack: local variables, function arguments, return addresses, temporary storage
- Heap: dynamically allocated memory
- Global/static: global variables, static variables
- Code: program instructions

Memory Layout



The Stack

- In C, all variables local to a function and function arguments are stored on the stack
- To call a function the code does:

  push arguments onto stack

  push return address onto stack

  jump to function code

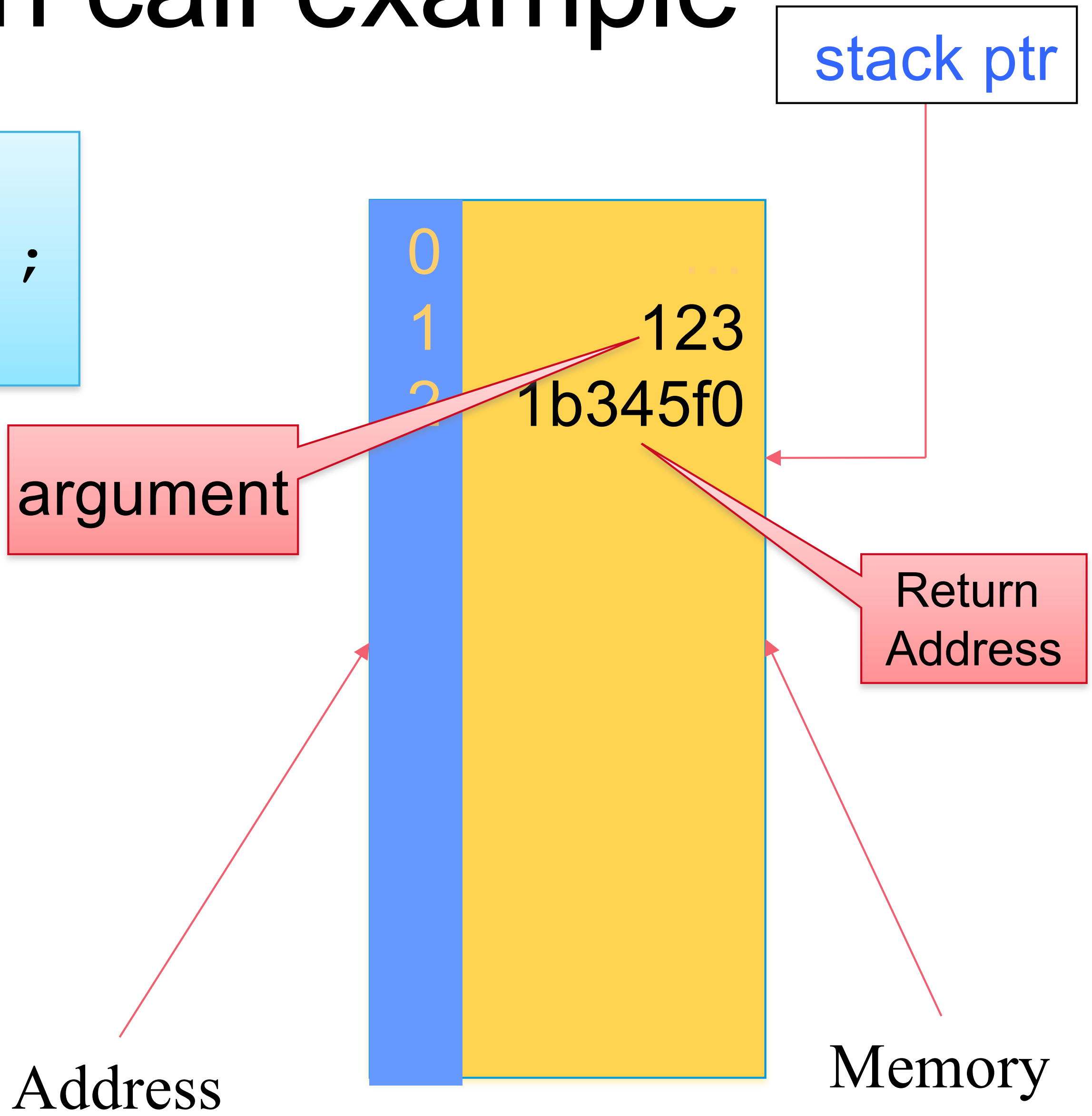
The Stack

- Inside the function, the code does the following:

  increment the stack pointer to allow
  space for the local variables
  execute the code
  pop local variables and arguments off the stack
  push the return result onto the stack
  jump to return address

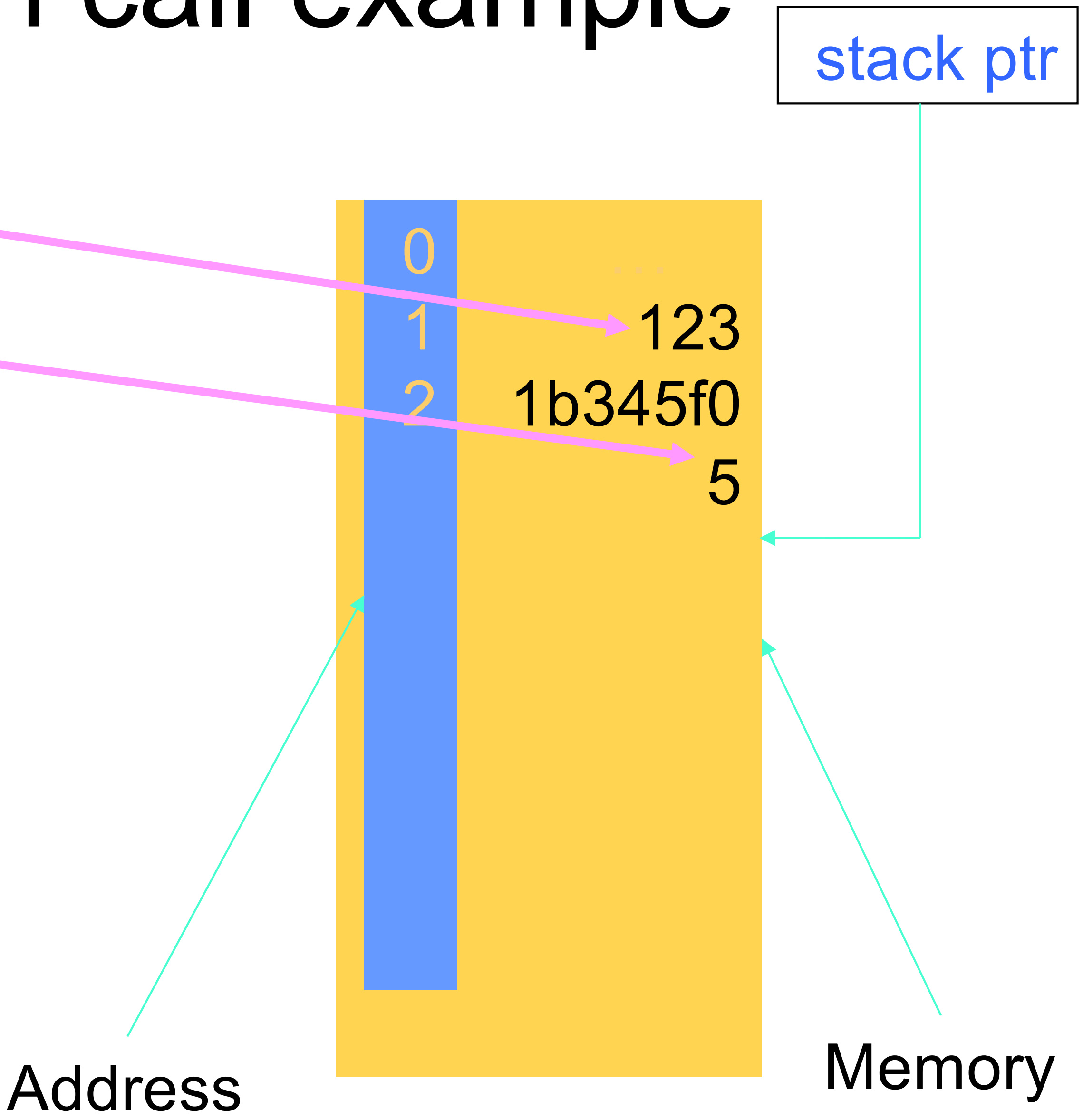
Function call example

```
. . .
res = myfun(123);
. . .
```



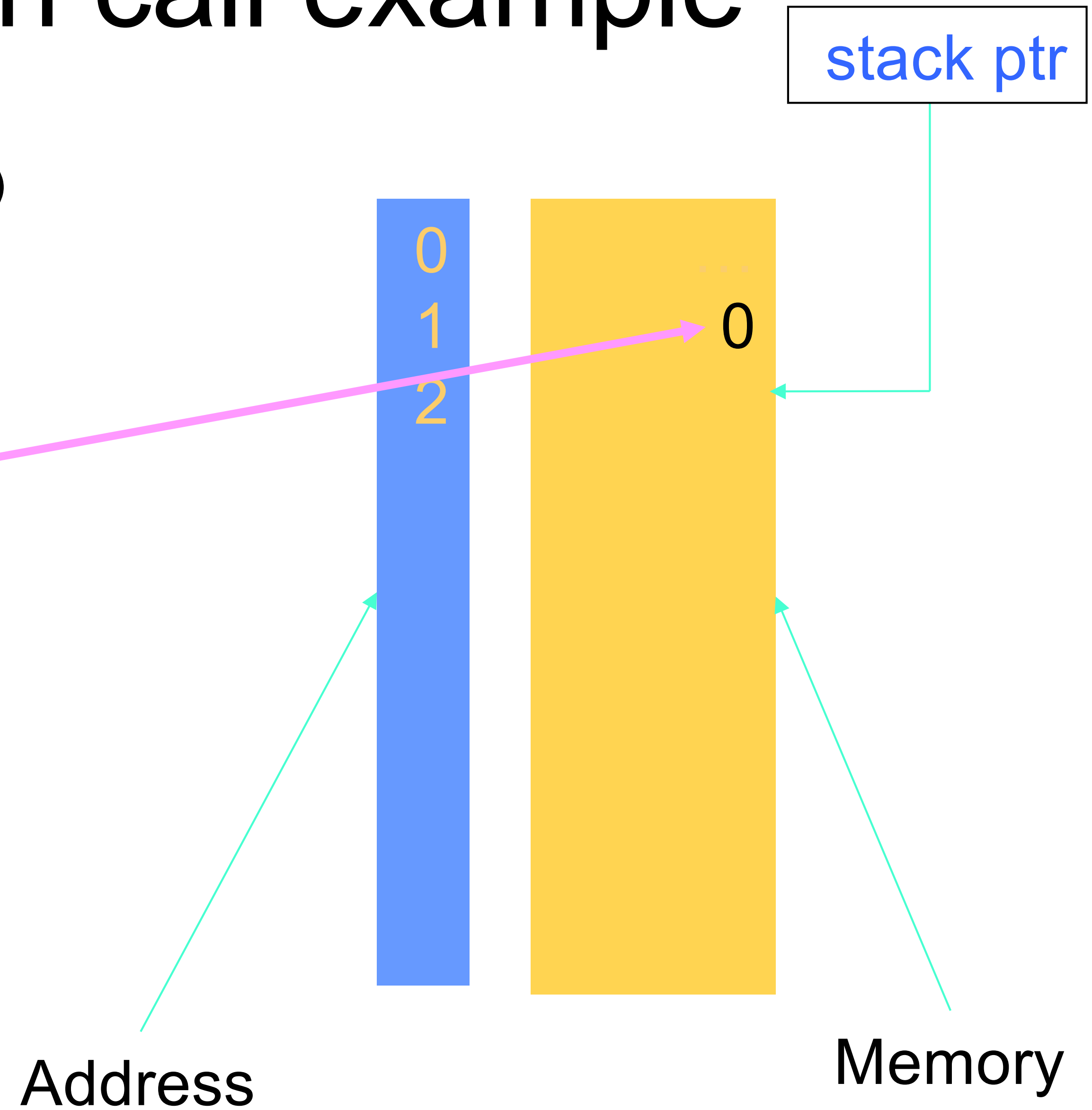
Function call example

```
int myfun(int a)
{
        int b = 5;
        …
        return 0;
}
```



Function call example

```
int myfun(int a)
{
        int b = 5;
        …
        return 0;
}
```



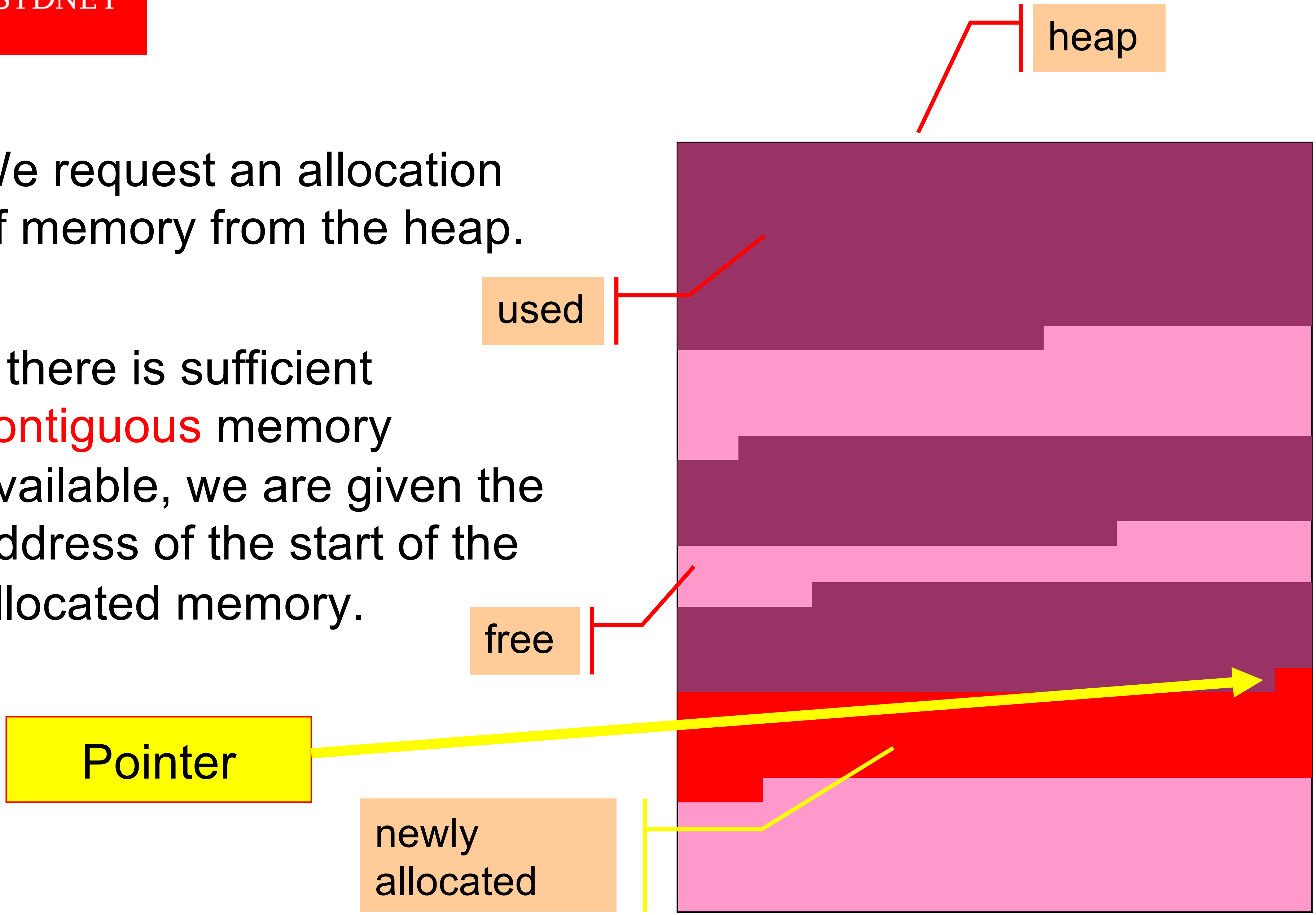
Heap

Memory may be dynamically allocated at run-time from an area known as "the heap".

Unlike the stack, which meets the temporary storage demands associated with called functions, the heap is accessed under direct programmer control.

We request an allocation of memory from the heap.

If there is sufficient contiguous memory available, we are given the address of the start of the allocated memory.



THE UNIVERSITY OF SYDNEY

Q: Where are parts of this program stored?

```
int a;
int main() {
    int b;
    int *p;
    p = malloc(…)
}
int doit(int c) {
    static int d;
}
```

```
int a;
int main() {
    int b;
    int *p;
    p = malloc(…)
}
int doit(int c) {
    static int d;
}
```

Q: What is the following *Java* code doing?

```
myObject fred = new   myObject();
```

A: Creating an object of type *myObject*.

However, what you *don't* see is the memory allocation required to instantiate the object.

Java also hides the act of freeing memory via automatic "garbage collection".

SUMMARY

Memory allocation is ***not*** difficult!

It only causes problems because novice programmers may not recognise the need to address it...

Java programmers are less likely to experience such problems simply because *Java* hides the need to deal with this whole issue.

Memory Management Functions

Memory allocation functions

Memory allocation functions return a "pointer to void".

A "pointer to void" is used to represent a pointer with no scalar value.

The pointer must therefore be cast to a specific type.

Memory allocation functions: malloc

```
#include <stdlib.h>
void *malloc(size_t size);
```

Typically defined as:
`typedef unsigned int size_t;`

Requests **`size`** number of bytes of memory.

Returns a pointer to the allocated memory, if successful, or a NULL pointer if unsuccessful

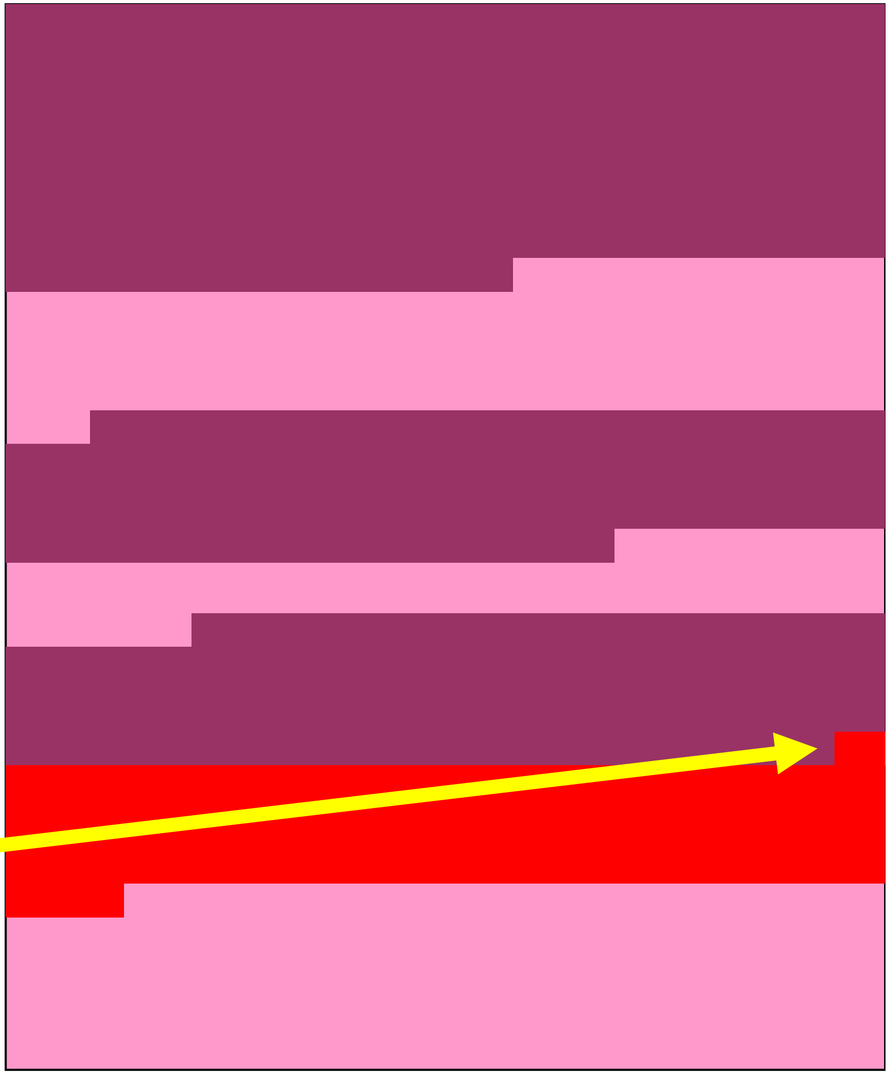
A comment on the use of `size_t`:

Use of `size_t` replaces the use of more specific types, such as `int`, `short`, etc. This allows the actual implementation to be system-specific.

The `sizeof` operator is of type `size_t`. This is often used to specify memory requirements, so it makes sense to have the size argument in memory allocation functions of type `size_t`.

```c
int *  ptr;
ptr = (int *)malloc(sizeof(int)*20);
```

If an **int** is 4 bytes, then this call will request 80 bytes of memory from the heap.



calloc

```
#include <stdlib.h>
void *calloc(size_t num, size_t size);
```

This is similar to **`malloc`** except that:

- It has two arguments:
  - **`num`** specifies the number of "blocks" of contiguous memory
  - **`size`** specifies the size of each block
- The allocated memory is cleared (set to '0').

free

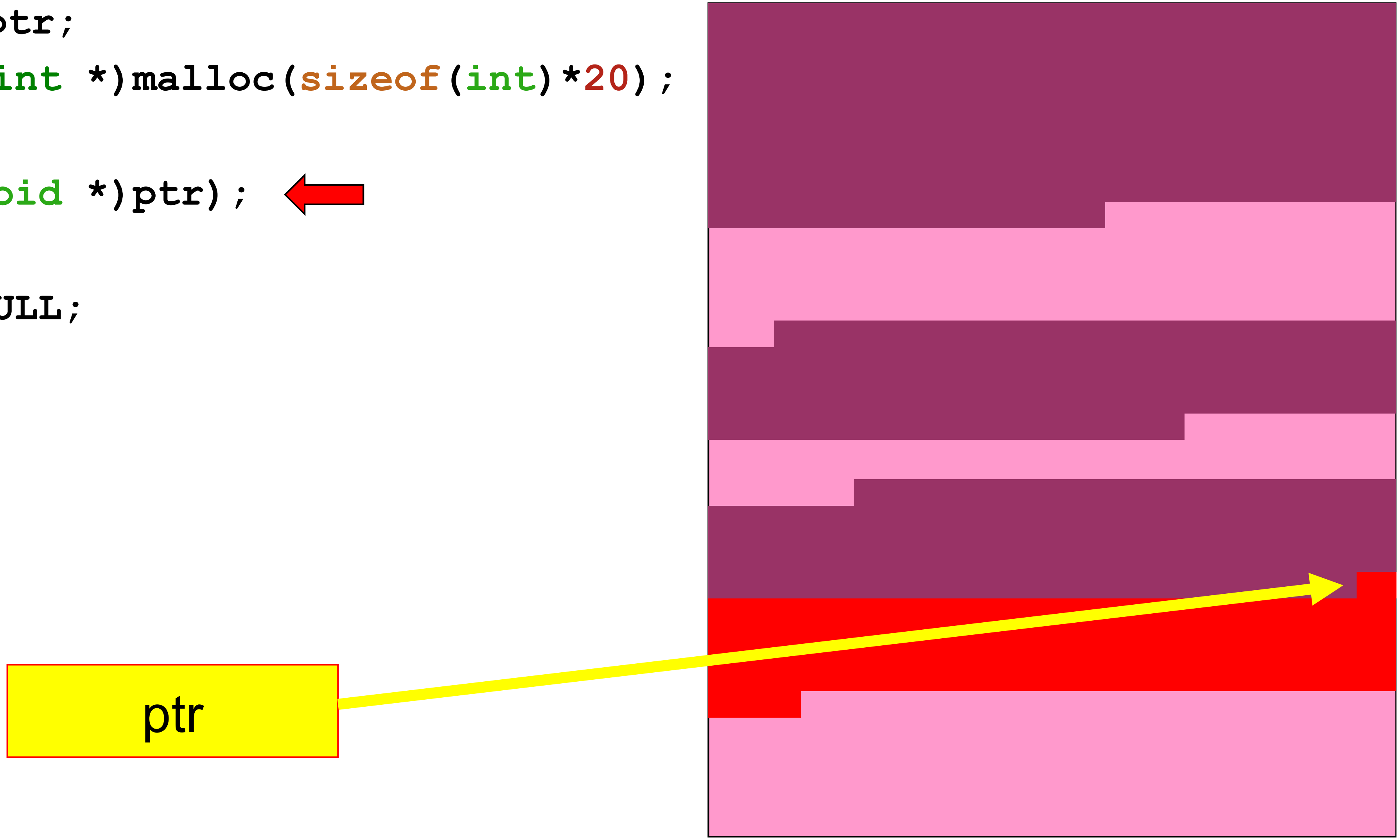
```
#include <stdlib.h>
void free(void *ptr);
```

This is used to de-allocate memory previously allocated by any of the memory allocation functions.

```c
int *  ptr;
ptr = (int *)malloc(sizeof(int)*20);

free((void *)ptr);

Ptr = NULL;
```



realloc

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

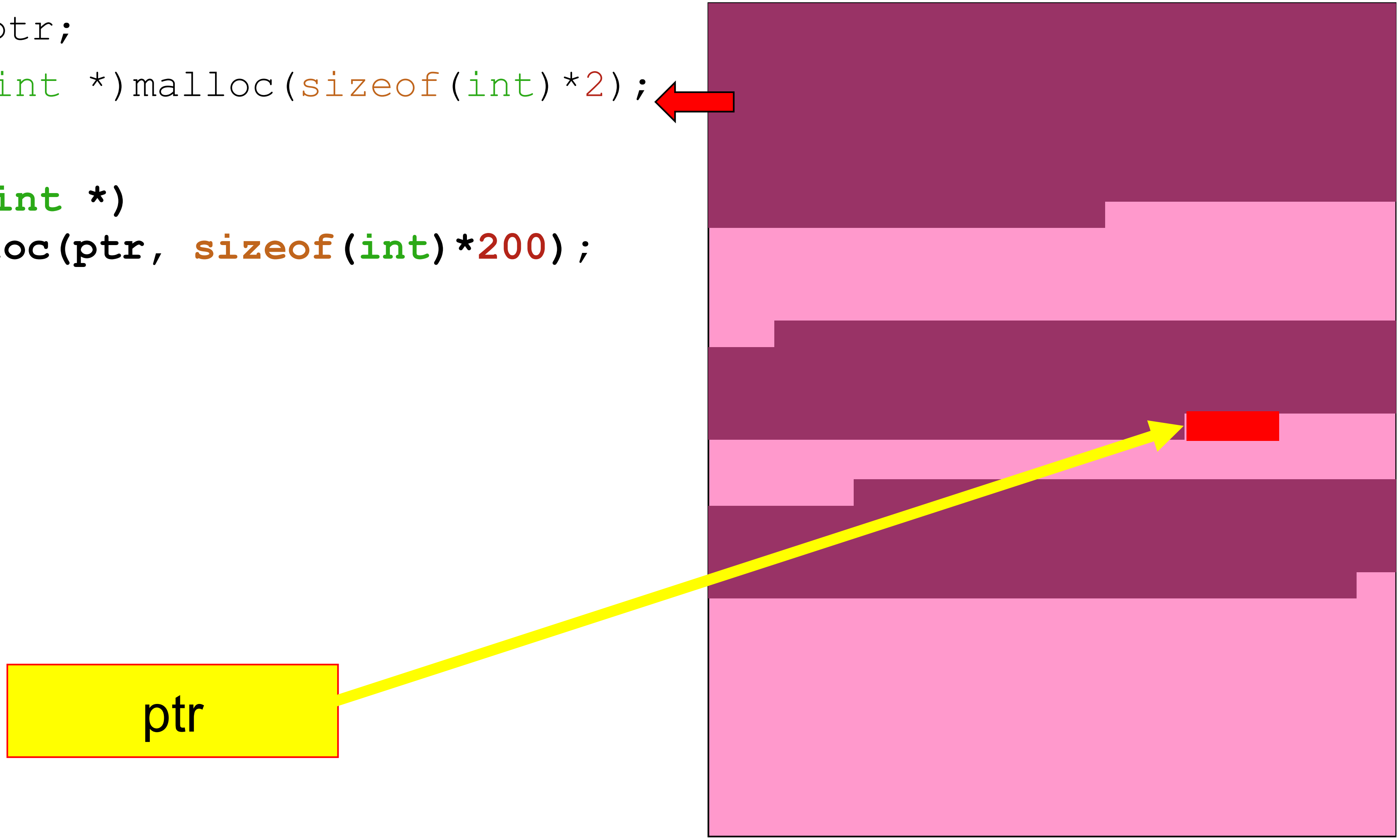
This takes previously-allocated memory and attempts to resize it.

This may require a new block of memory to be found, so it returns a new void pointer to memory.

Contents are preserved.

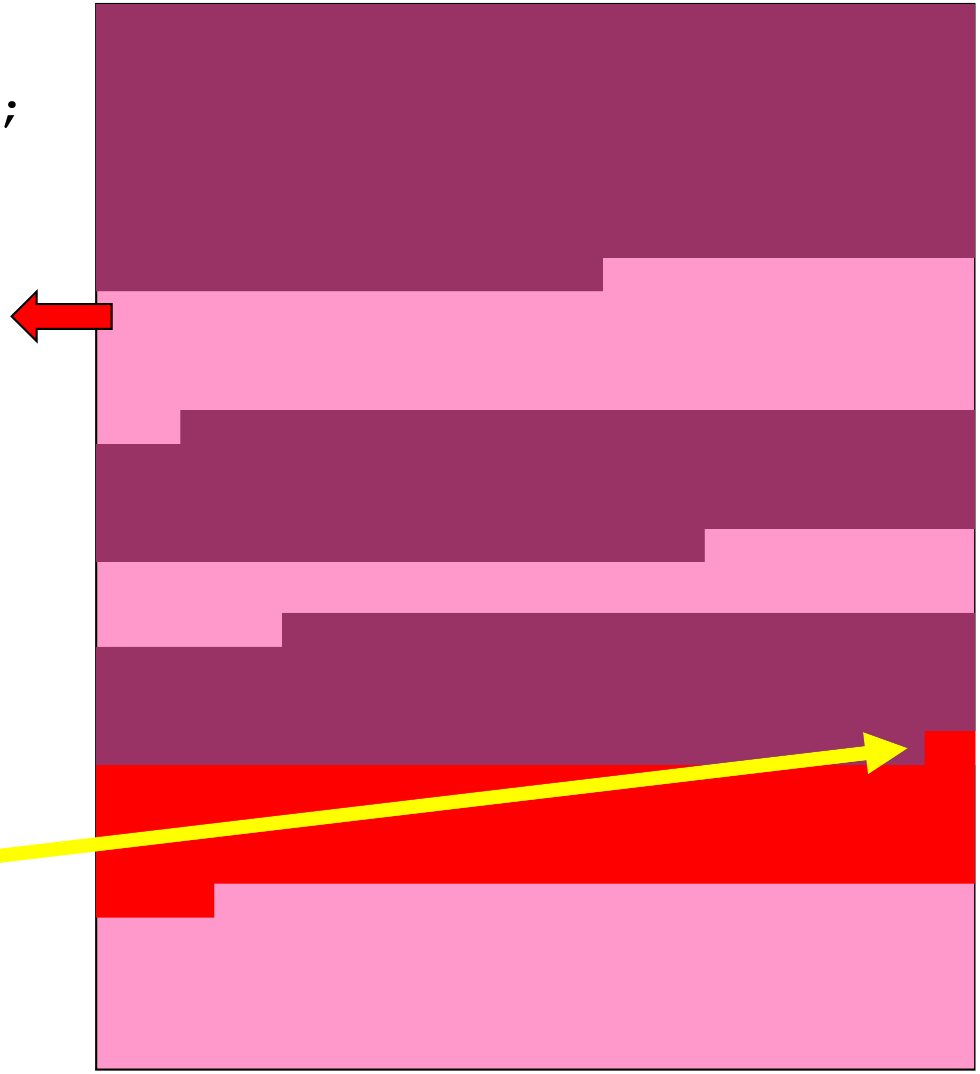
```
int *  ptr;
ptr = (int *)malloc(sizeof(int)*2);

ptr = (int *)
   realloc(ptr, sizeof(int)*200);
```



```
int *  ptr;
ptr = (int *)malloc(sizeof(int)*2);

ptr = (int *)
   realloc(ptr, sizeof(int)*200);
```



Dynamically creating structures

```
struct thing *    ptr;

ptr = (struct thing *)malloc(sizeof(struct thing));

/* Do stuff */
ptr->day = mon;
 …
free((void *)ptr);
Ptr = NULL;
```

This is a some of what *Java* does "behind the scenes" on object creation.

Safety issues

Safety issues

Caution #1:

De-allocate memory that is no longer required.

While the system should de-allocate resources on termination, it is good practice to take control of this process.

In some *Java* programs there is a noticeable performance dip when the automatic "garbage collection" functionality kicks in.

Safety issues

Caution #2:

NEVER attempt to de-allocate memory that has not been allocated!

A common error is to try to free memory that has already been de-allocated, or was never allocated in the first instance.

Safety issues

Caution #3:

NEVER try to use memory that has been de-allocated.

This is also a common error leading to serious problems.

Safety issues

Caution #4:

Know your memory allocation requirements!

Use of the **`sizeof`** operator addresses the more obvious problems.

However, a common problem is to forget that a string includes a '\0' terminating character.

Safety issues

Caution #5:

Check for success!

A failed memory allocation request can lead to disaster if it is simply assumed to be successful.

Previous examples here have made this assumption for convenience. This would NOT qualify as bullet-proof code!

Safety issues

Typically, safe memory allocation is addressed by wrapping the relevant function in some additional code.

The following code* demonstrates an example using `realloc`.

* Adapted from Kay & Kummerfeld, *C Programming in a UNIX environment*

Safety issues

```
#include <stdlib.h>

void *
srealloc(void *ptr, size_t size)
{
  void *res;
  if((res = realloc(ptr, size)) == (void *)0)
  {
      perror("realloc()");
      exit(1);
  }
  return res;
}
```

If the returned result is a NULL pointer, let the system print the appropriate error message via **perror** and then **exit**.

Otherwise, return the pointer to memory.

Safety issues

- **salloc**
- **srealloc**

- **Test return value on malloc, realloc**

Summary

- ✓ Understand the need for memory allocation and de-allocation
- ✓ Be able to use relevant *C* functions for achieving this
  - ✓ malloc
  - ✓ calloc
  - ✓ realloc
  - ✓ free
- ✓ Be able to allocate and access memory *safely*

Sources

- Image sources:
  - zazzle t-shirt
  - http://www.hazoment.com/Humor-Fasten_Safety_Belts.jpg

- Kay, J. & B. Kummerfeld (1989). *C Programming in the UNIX environment.* Addison-Wesley: Sydney.