# Introduction to Dynamic Memory Management

FACULTY OF
ENGINEERING &
INFORMATION
TECHNOLOGIES

COMP2129

THE UNIVERSITY OF
SYDNEY

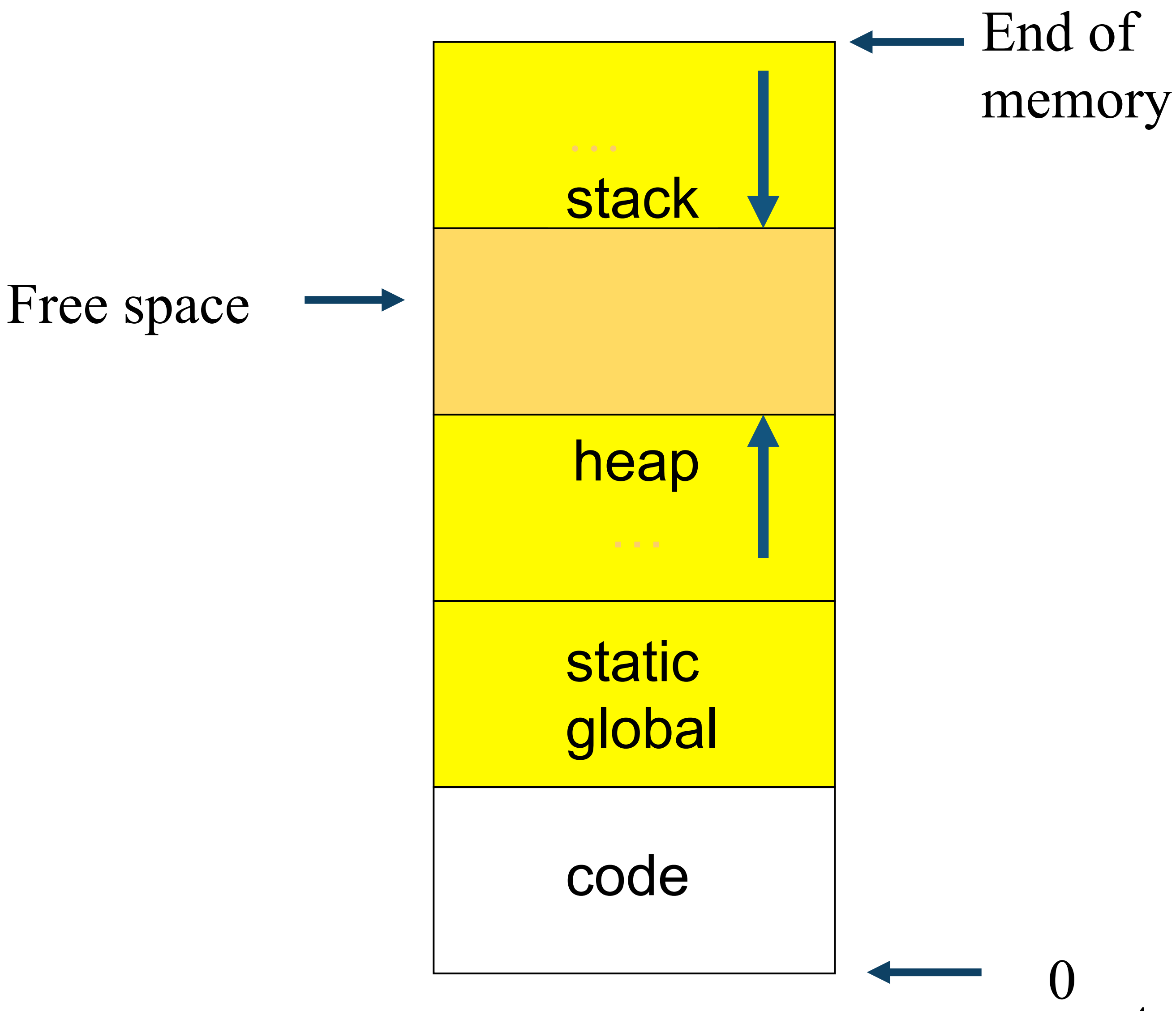
# Memory

- 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

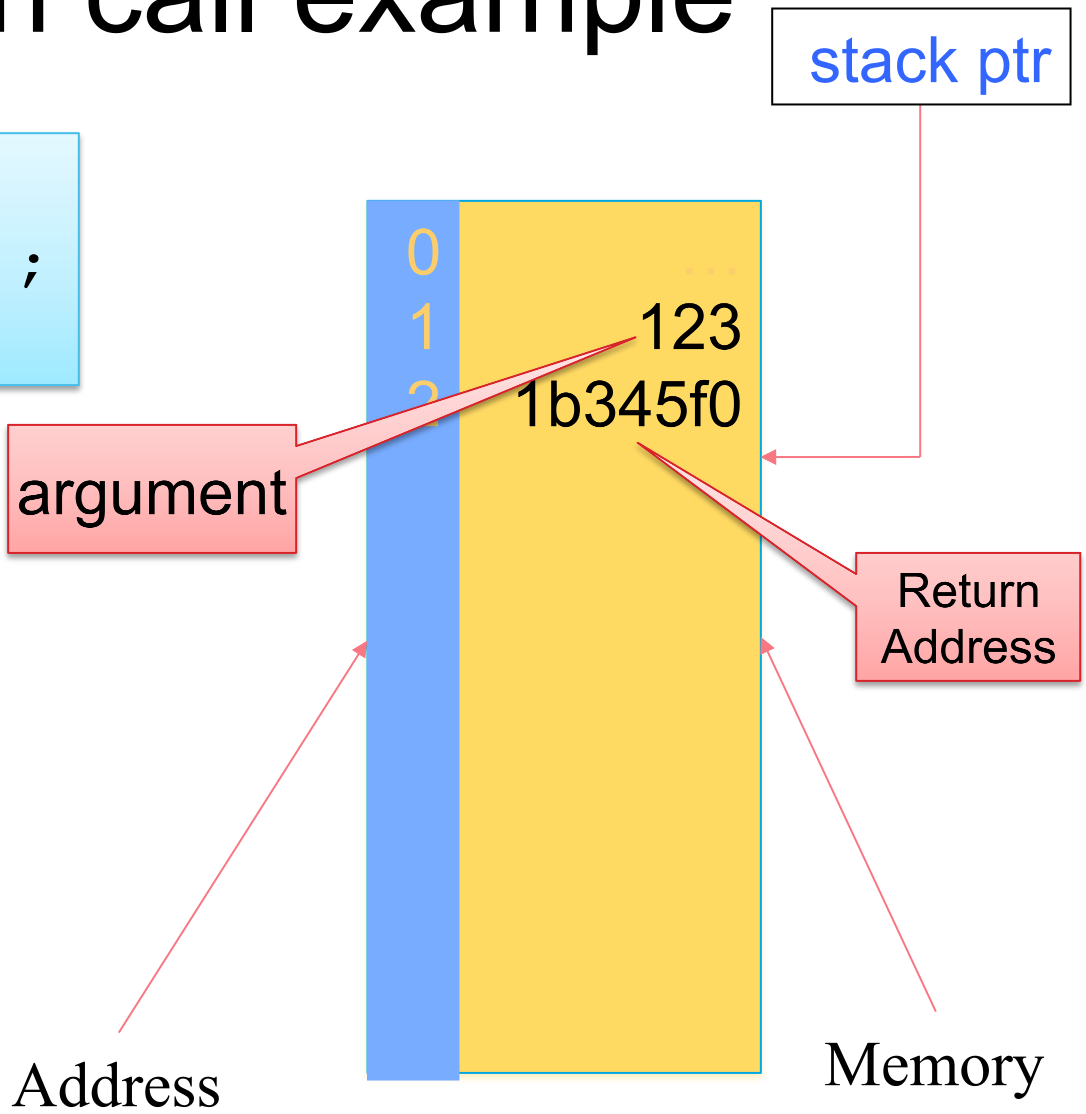
# The Stack

- 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

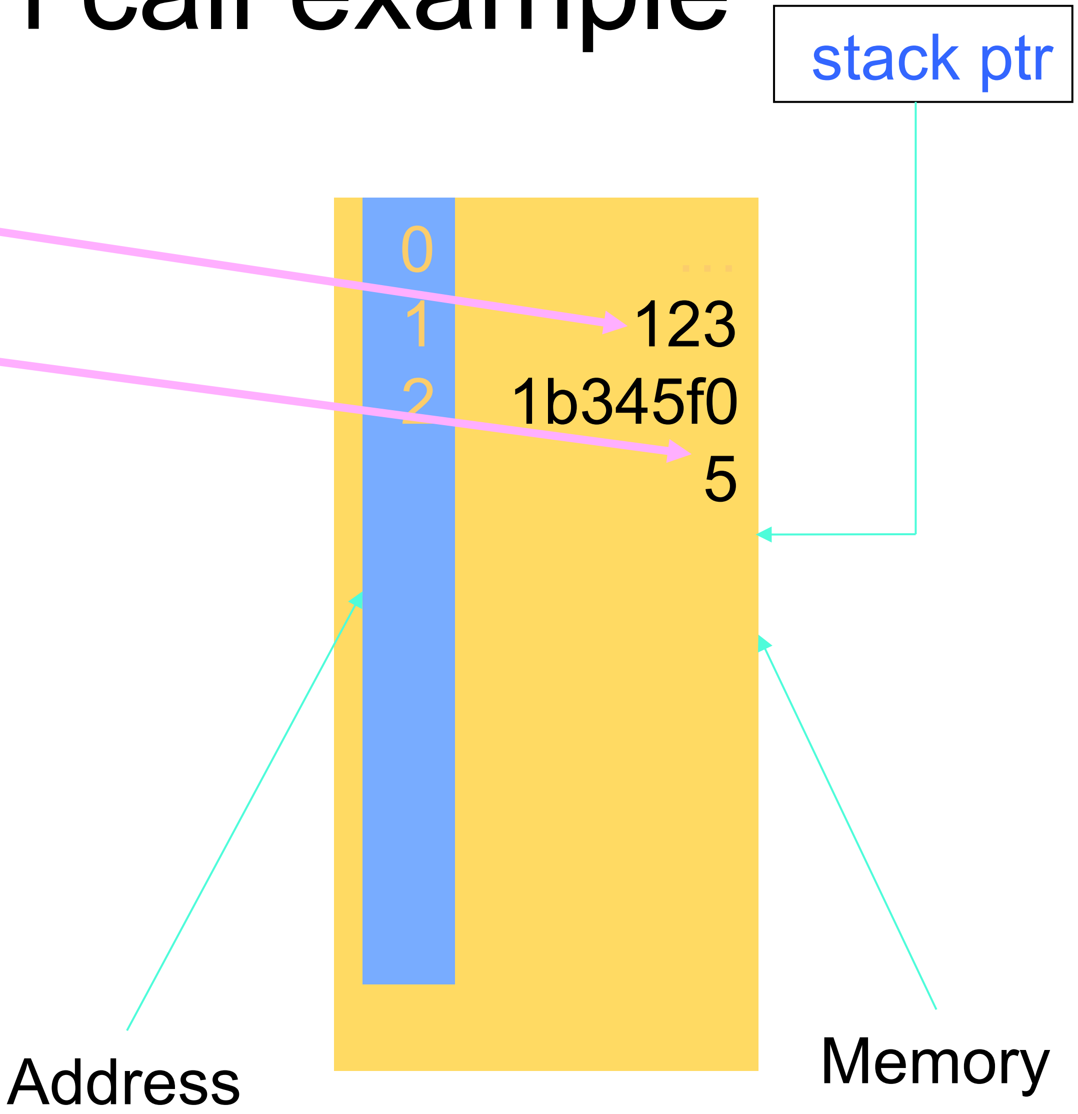
# Function call example

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



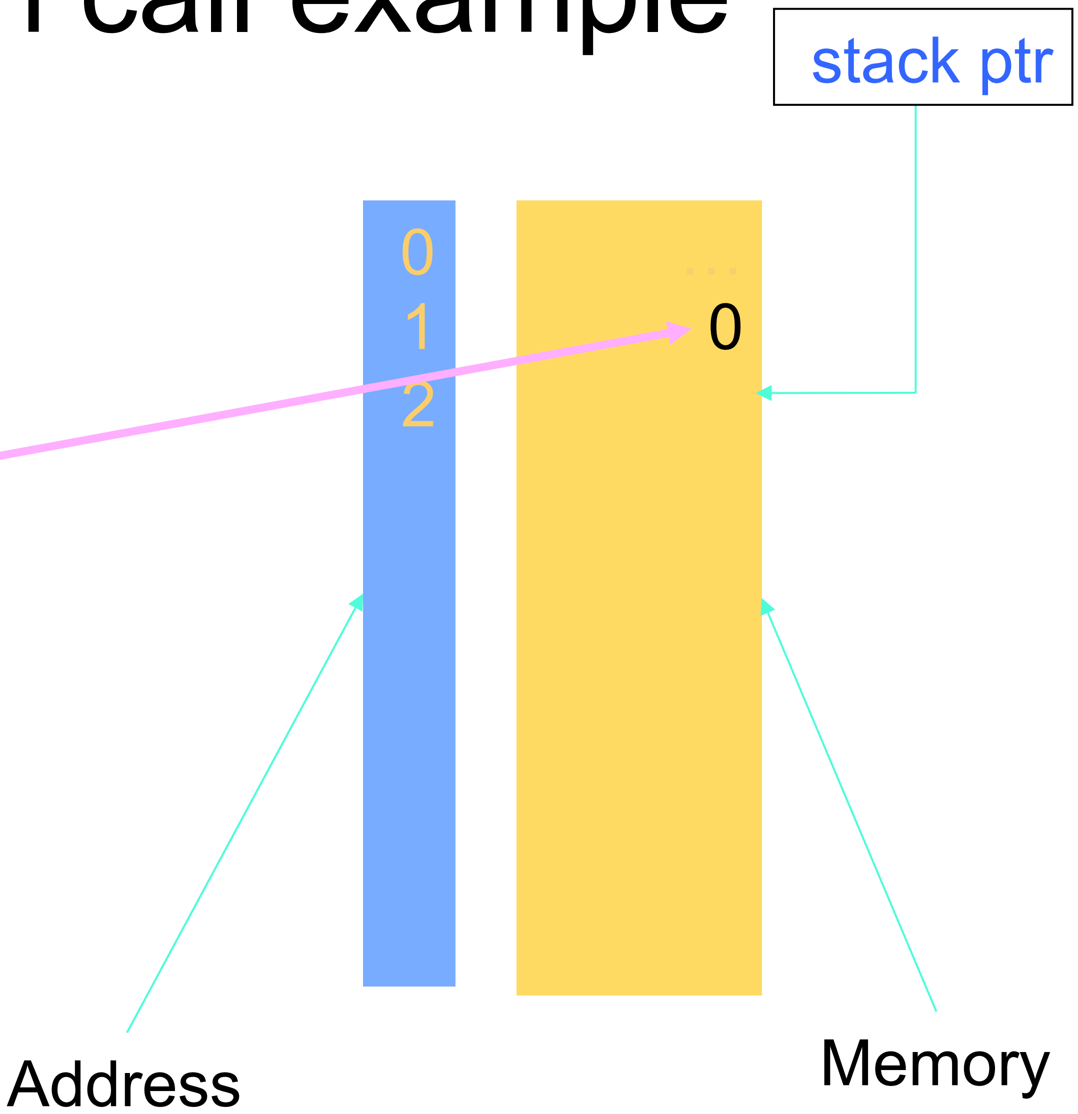
# Function call example

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



# Function call example

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



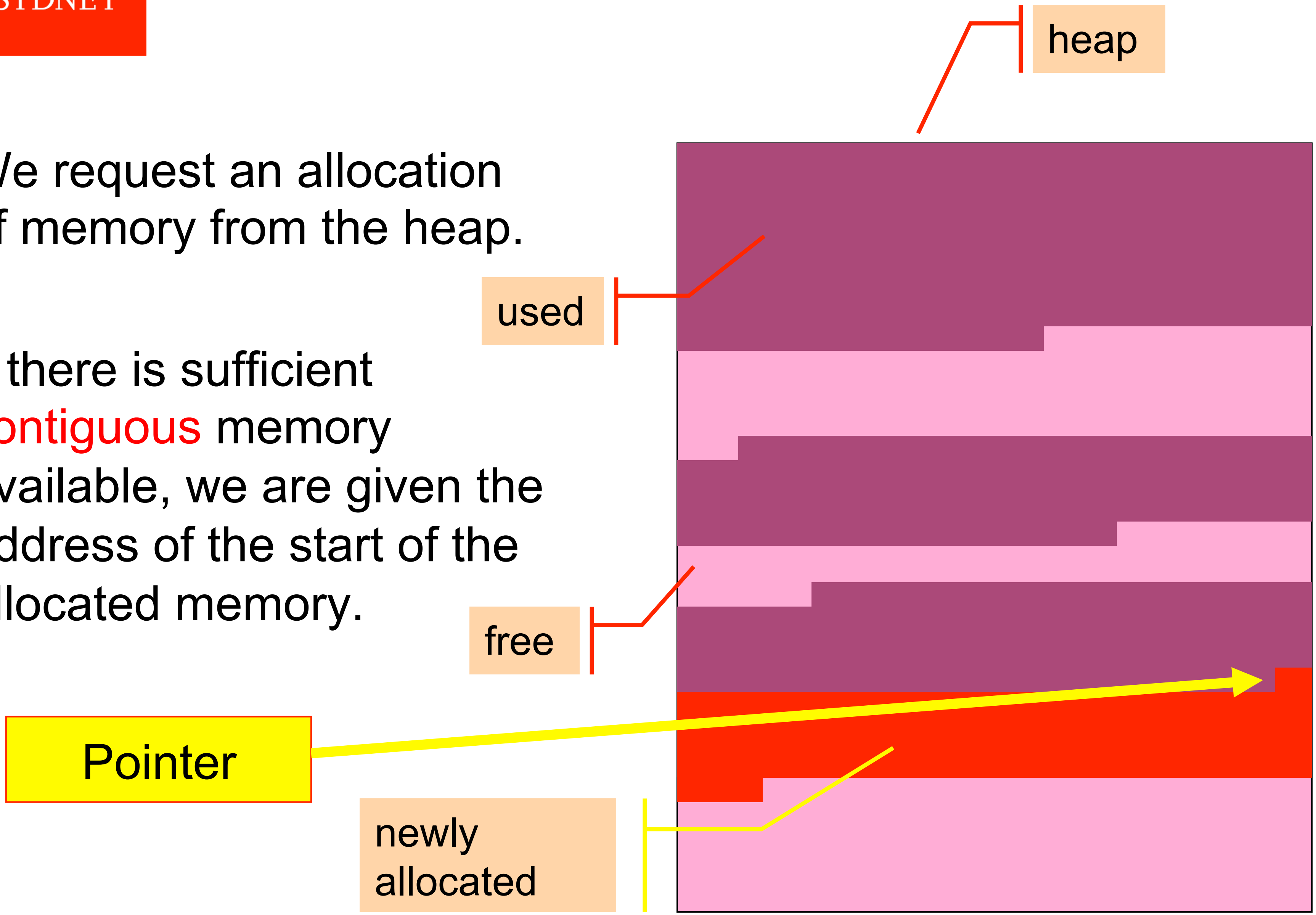
# Heap

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.



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;
}
```

THE UNIVERSITY OF SYDNEY

**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

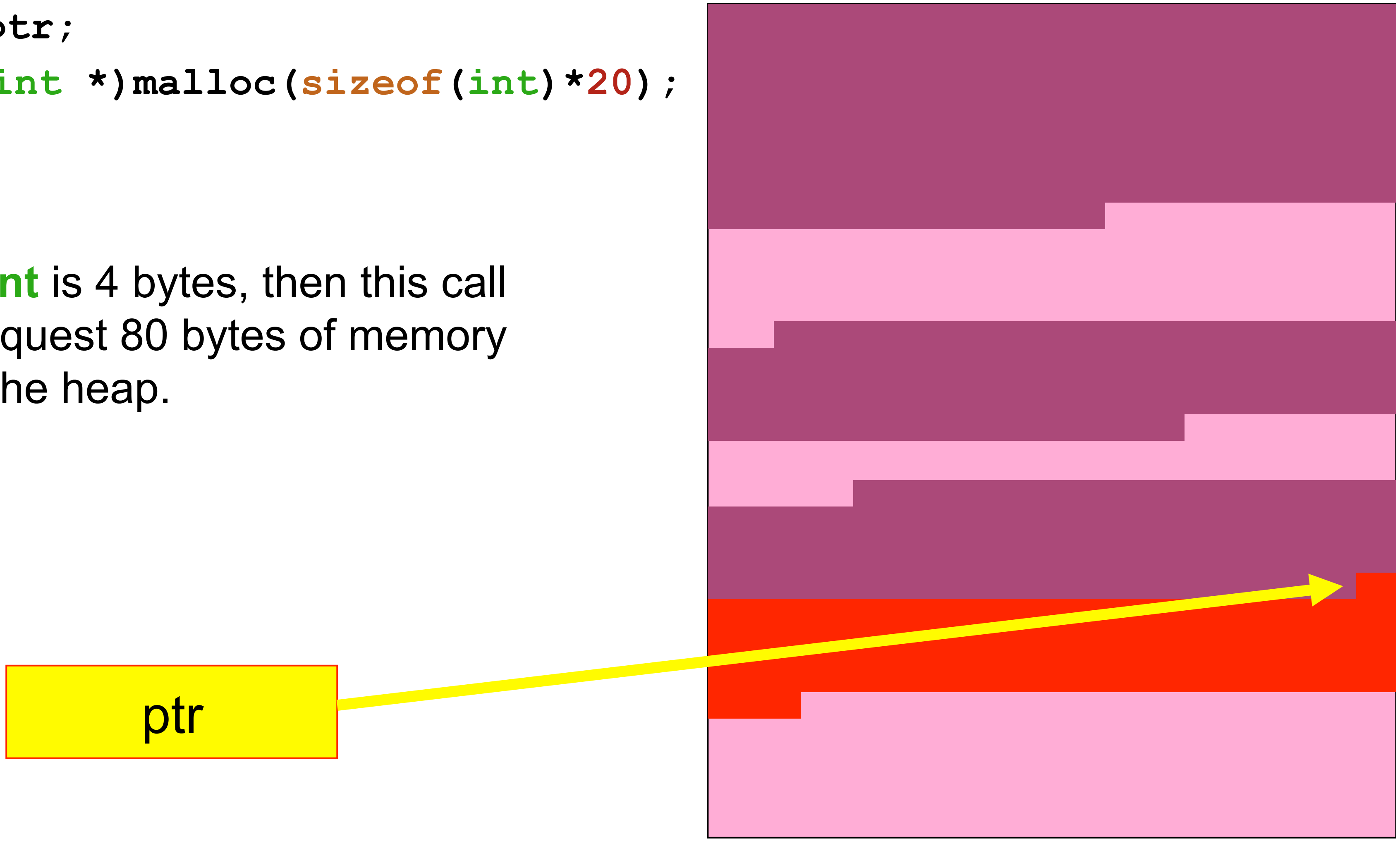
A comment on the use of `size_t`:

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’).

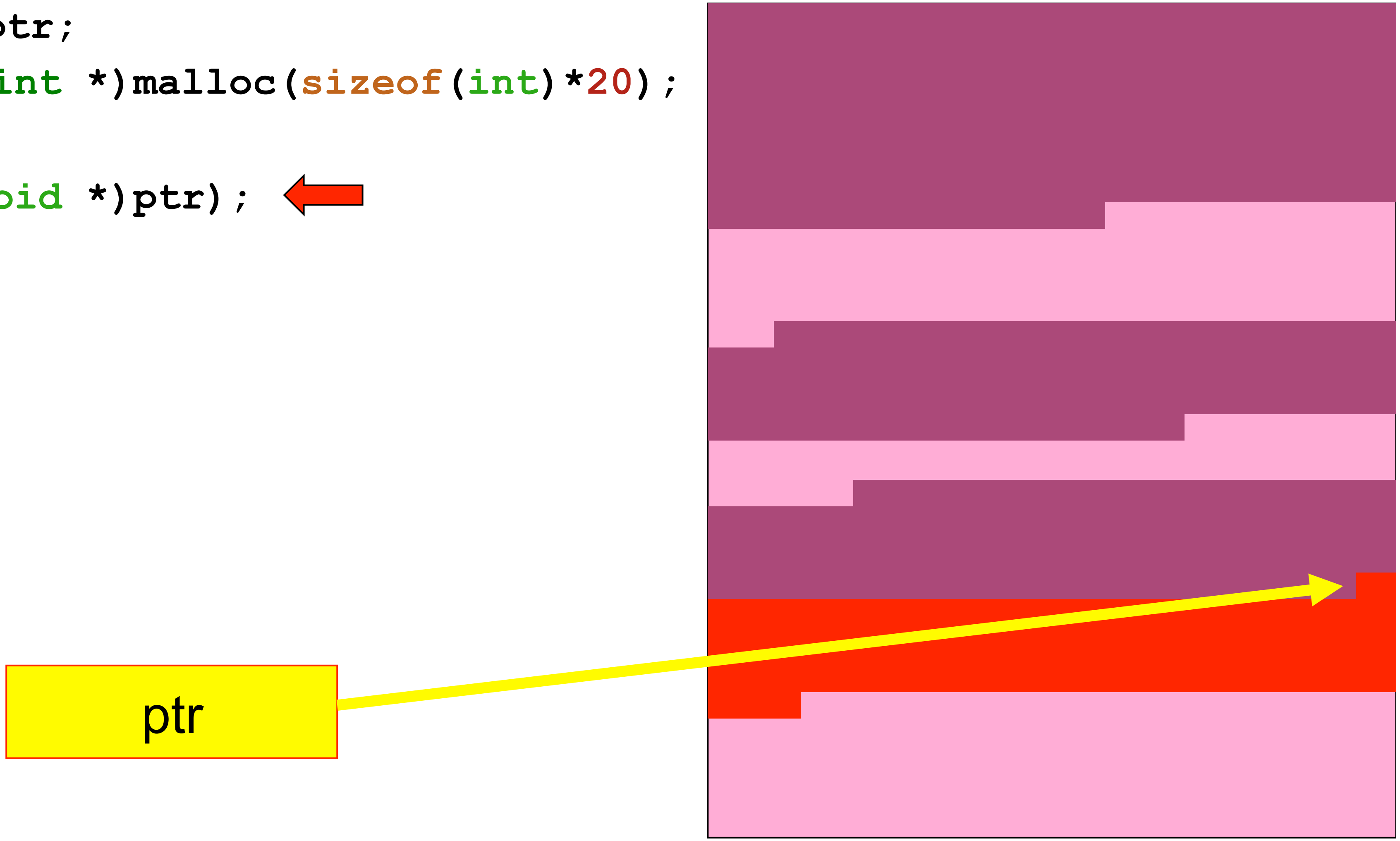
# free

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

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

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

free((void *)ptr);
```



# realloc

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

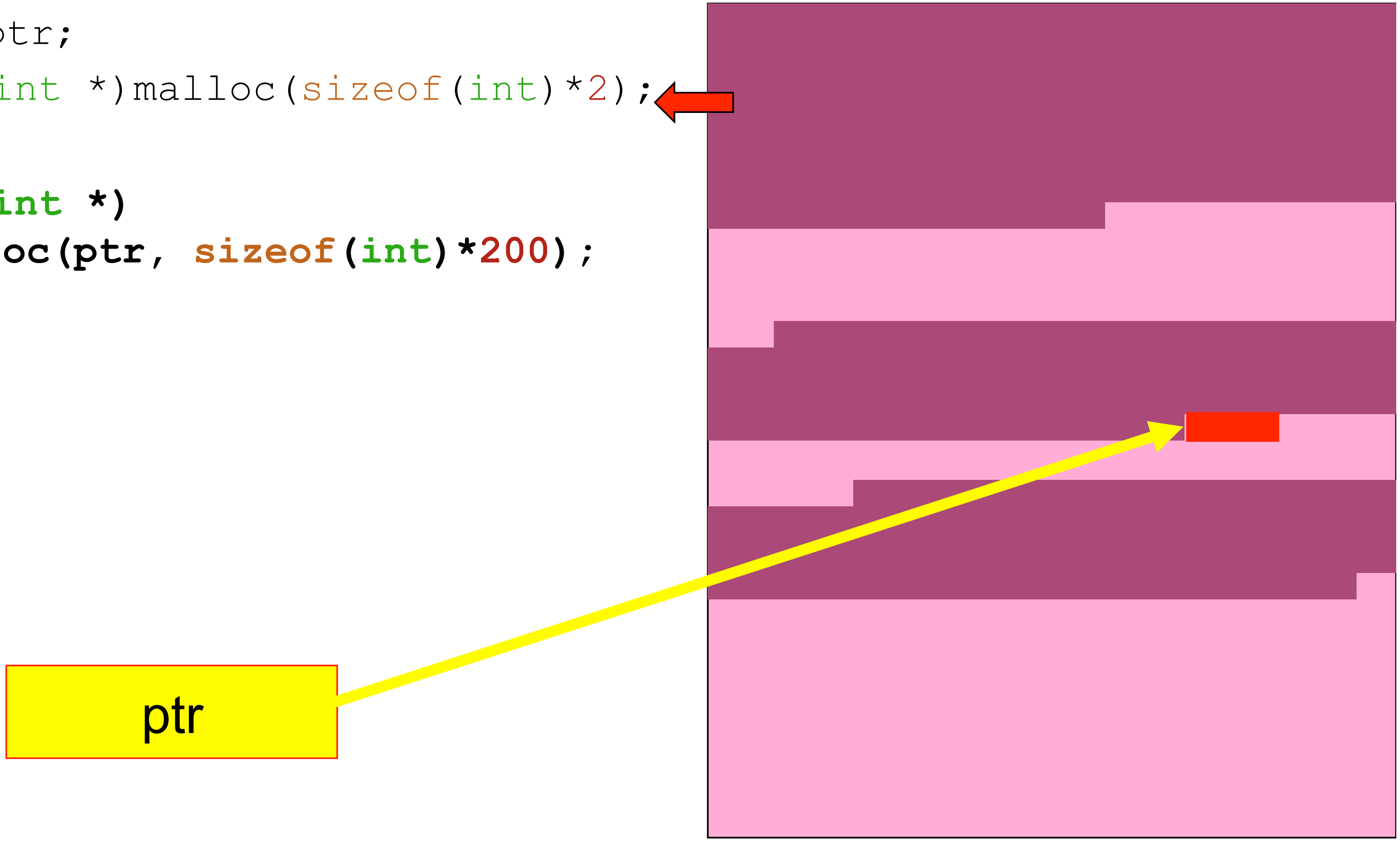
This takes previously-allocated memory and attempts to resize it.

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

Contents are preserved.

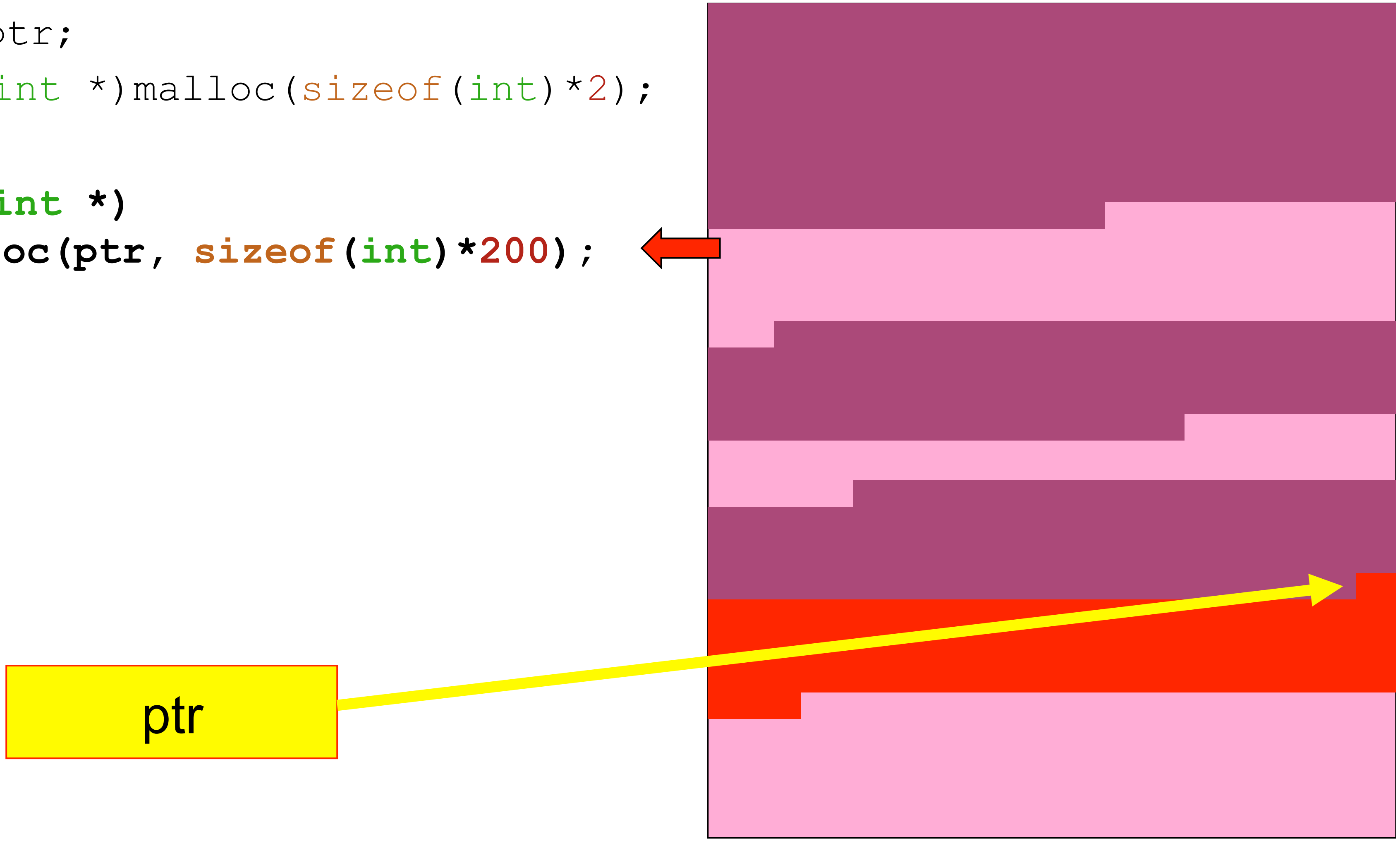
```c
int *  ptr;
ptr = (int *)malloc(sizeof(int)*2);

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



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

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



THE UNIVERSITY OF SYDNEY

# Dynamically creating structures

```
struct thing *    ptr;

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

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

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 NULL 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.

# Software development and debugging

FACULTY OF ENGINEERING & INFORMATION TECHNOLOGIES

COMP2129

THE UNIVERSITY OF SYDNEY

# Software development is a process

An example

1. Gather requirements in human terms
2. Identify information: input/output/stored/processed etc.
3. Describe program operation as a set of design document(s)
4. Create as many test cases as possible and check against design document(s), when it doesn't work goto 1
5. Write up implementation based on design document
6. Make more tests based on implementation specifics
7. Run tests,
    1. Test fail -> debug, goto 5
    2. Test mega fail -> goto 1

# Text anonymiser

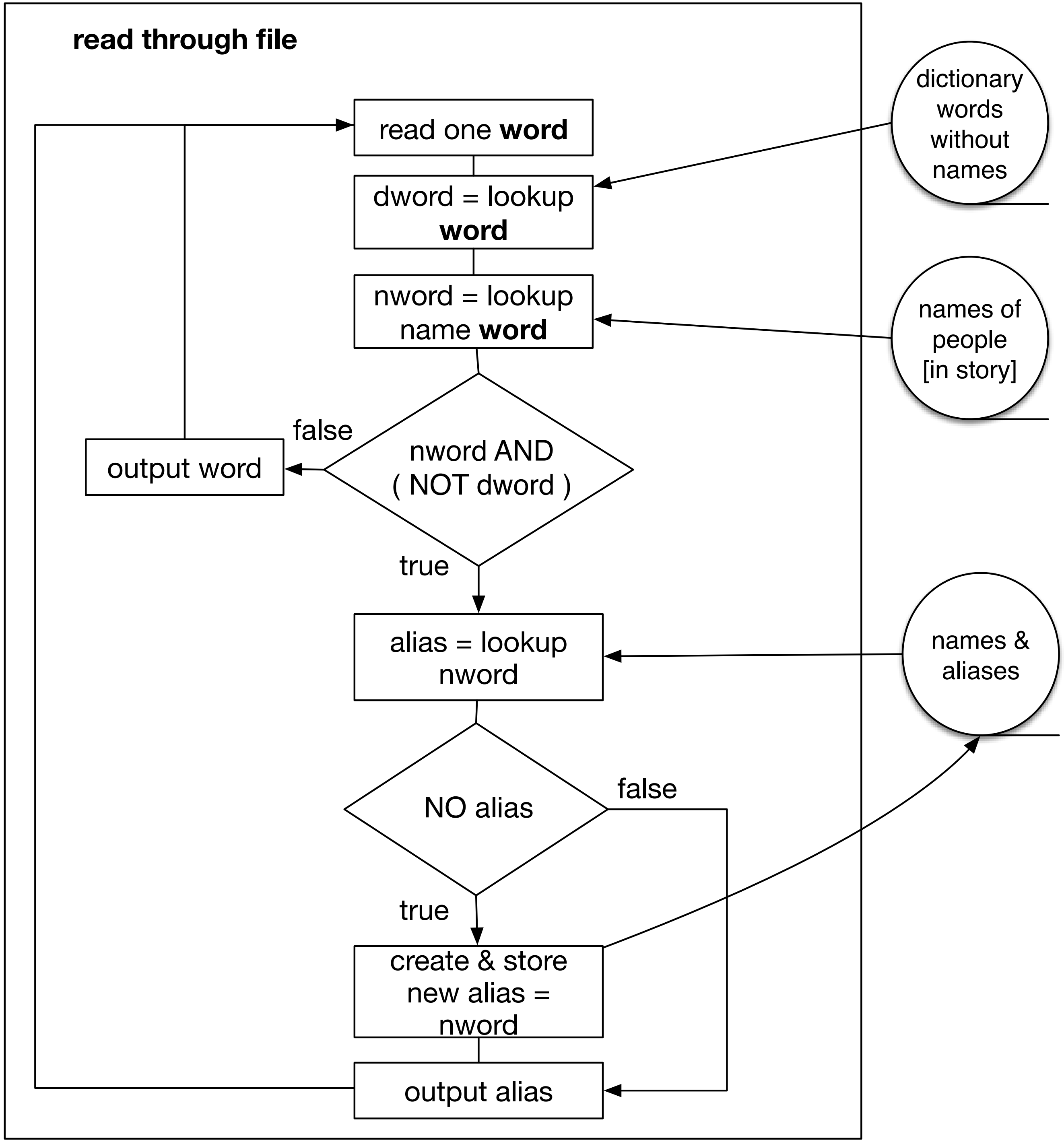
Protect identity of real persons from a text file

Normally, a person will read through the document and swap the name of the person with an acronym that is in no way related, e.g. Mr. Barry Higgins becomes CK.

The name will appear in different forms and will need to be replaced with the acronym when there is reference made.

Requirements?

Information inputs/outputs/processing/storage etc.


read through file
read one word
dword = lookup word
nword = lookup name word
nword AND ( NOT dword )
false
true
output word
alias = lookup nword
NO alias
false
true
create & store new alias = nword
output alias
dictionary words without names
names of people [in story]
names & aliases

# Fast forward to Debugging

- Debugging should not be fixing program logic
  - Debugging should be fixing program errors

- Print statement are not debugging

- A debugging tool will help identify the errors in the implementation
  - Typos,  limits of types, loop too much/little

# Example

```c
#include <stdio.h>
#include <math.h>

int main()
{
    int x, y[10];
    for (x = 5; x >= -5; x--) {
        y[x+5] = 1 / x ;
    }
    for (x = 0; x < 10; x++) {
        printf("y[%d] = %d\n", x, y[x]);
    }

    printf("done\n");
    return 0;
}
```

# Example

```
int x, y[10];
  for (x = 5; x >= -5; x--) {
     y[x+5] = 1 / x ;
  }
  for (x = 0; x < 10; x++) {
     printf("y[%d] = %d\n", x, y[x]);
  }

  printf("done\n");
```

```
$ gcc feval.c -o feval
$ ./feval
$ Floating point exception: 8
$
```

# Example

```
#include <stdio.h>
#include <math.h>

int main()
{
    int x, y[10];
    for (x = 5; x >= -5; x--) {
        y[x+5] = 1 / x ;
    }
    for (x = 0; x < 10; x++) {
        printf("y[%d] = %d\n", x, y[x]);
    }

    printf("done\n");
    return 0;
}
```

# Enter debugger *db (gdb, lldb, jdb, ...

```
$ gcc -g feval.c -o feval
$ lldb feval
(lldb) target create "feval"
Current executable set to 'feval' (x86_64).
(lldb) run
Process 29413 launched: '/Users/homesauce/feval' (x86_64)
Process 29413 stopped
* thread #1: tid = 0x331d89, 0x0000000100000eb7 feval`main +
55 at feval:8, queue = 'com.apple.main-thread', stop reason =
EXC_ARITHMETIC (code=EXC_I386_DIV, subcode=0x0)
    frame #0: 0x0000000100000eb7 feval`main + 55 at feval.c:8
   5         {
   6                 int x, y[10];
   7                 for (x = 5; x >= -5; x--) {
-> 8                         y[x+5] = 1 / x ;
   9                 }
   10                for (x = 0; x < 10; x++) {
   11                        printf("y[%d] = %d\n", x, y[x]);
(lldb)
```

# Enter debugger *db (gdb, lldb, jdb, ...

```
(lldb) print x
(int) $1 = 0
(lldb) print y
(int [10]) $0 = ([0] = 0, [1] = 0, [2] = 0, [3] = 0, [4] = 0,
[5] = 0, [6] = 1, [7] = 0, [8] = 0, [9] = 0)
```

THE UNIVERSITY OF SYDNEY

# Example

```
#include <stdio.h>
#include <limits.h>
#include <math.h>

int main() {
   int x, y[10];
   for (x = 5; x >= -5; x--) {
      if (x == 0) {
            y[x+5] = INT_MAX;
            continue;
       }
       y[x+5] = 1 / x ;
   }
   for (x = 0; x < 10; x++) {
      printf("y[%d] = %d\n", x, y[x]);
   }
   printf("done\n");
   return 0;
}
```

# Enter debugger *db (gdb, lldb, jdb, ...

```
(lldb) breakpoint set --file feval.c --line 7
Breakpoint 1: where = feval`main + 29 at feval.c:7, address =
0x0000000100000e7d
(lldb) run
…
    frame #0: 0x0000000100000e7d feval`main + 29 at feval.c:7
   4
   5    int main() {
   6        int x, y[10];
-> 7        for (x = 5; x >= -5; x--) {
   8            if (x == 0) {
   9                     y[x+5] = INT_MAX;
   10                    continue;
(lldb) print x
(int) $0 = 0
(lldb) si
…
(lldb) p x
(int) $1 = 5
(lldb) bt
* frame #0: 0x0000000100000e7d feval`main + 29 at feval.c:7
  frame #1: 0x00007fff93f4e5c9 libdyld.dylib`start + 1
  frame #2: 0x00007fff93f4e5c9 libdyld.dylib`start + 1
```

(br set -f feval.c -l 7)

Step Into (si): goto next instruction

print (p)

bt: print backtrace (stack of function calls)

# Fast forward to Debugging

- Set breakpoints in area of interest
- Run
- Print memory contents
- Do some stepping
  - Instruction level and/or
  - Source code level
- Print memory contents
  - Change memory contents?
  - Change frame?
- Monitor specific variables with watchpoints

lldb or gdb

The choice is yours