Malloc and Free

Consider the bad code

```
a = (int*)malloc(n*sizeof(int));
a[7] = 42;
a = (int*)malloc(m*sizeof(int));
```

Malloc and Free

Consider the bad code

```
a = (int*)malloc(n*sizeof(int));
a[7] = 42;
a = (int*)malloc(m*sizeof(int));
```

Each `malloc` allocates a new chunk of memory

Malloc and Free

Consider the bad code

```
a = (int*)malloc(n*sizeof(int));
a[7] = 42;
a = (int*)malloc(m*sizeof(int));
```

Each `malloc` allocates a new chunk of memory

The first chunk is still allocated, but is no longer accessible by the program as it no longer knows where it is

Malloc and Free

Consider the bad code

```
a = (int*)malloc(n*sizeof(int));
a[7] = 42;
a = (int*)malloc(m*sizeof(int));
```

Each `malloc` allocates a new chunk of memory

The first chunk is still allocated, but is no longer accessible by the program as it no longer knows where it is

We have overwritten the address of the memory: it could have been anywhere, we don't know anymore

Malloc and Free

Consider the bad code

```
a = (int*)malloc(n*sizeof(int));
a[7] = 42;
a = (int*)malloc(m*sizeof(int));
```

Each `malloc` allocates a new chunk of memory

The first chunk is still allocated, but is no longer accessible by the program as it no longer knows where it is

We have overwritten the address of the memory: it could have been anywhere, we don't know anymore

That area of memory is now *garbage*. It takes up space but the program can't get at it

Malloc and Free

If we do this too much, then memory will fill up with inaccessible garbage, and we will probably run out

Malloc and Free

If we do this too much, then memory will fill up with inaccessible garbage, and we will probably run out

Of course, the correct thing is to call `free` on `a` before we overwrite it

Malloc and Free

If we do this too much, then memory will fill up with inaccessible garbage, and we will probably run out

Of course, the correct thing is to call `free` on `a` before we overwrite it

Or make a copy of the value of `a` somewhere else first

Malloc and Free

If we do this too much, then memory will fill up with inaccessible garbage, and we will probably run out

Of course, the correct thing is to call `free` on `a` before we overwrite it

Or make a copy of the value of `a` somewhere else first

The important thing is to ensure a pointer to every allocated chunk is somehow accessible (directly or indirectly) by the program and can be accessed or freed if necessary

Malloc and Free

Programs that create inaccessible areas of memory this (and there are many) are said to have a *memory leak*

Malloc and Free

Programs that create inaccessible areas of memory this (and there are many) are said to have a *memory leak*

Memory leaks often go unnoticed as programmers often test their programs on small examples: small enough that the amount of garbage is still small and `malloc` always succeeds

Malloc and Free

Programs that create inaccessible areas of memory this (and there are many) are said to have a *memory leak*

Memory leaks often go unnoticed as programmers often test their programs on small examples: small enough that the amount of garbage is still small and `malloc` always succeeds

They only discover the error when their code goes into production on big examples and then starts failing

Malloc and Free

Aside. Current operating systems clean up after you when your program exits, returning all malloced memory. Some early operating systems didn't, meaning poorly written programs could jam up the entire computer, eventually requiring a reboot

Malloc and Free

Aside. Current operating systems clean up after you when your program exits, returning all malloced memory. Some early operating systems didn't, meaning poorly written programs could jam up the entire computer, eventually requiring a reboot

Tools like `valgrind` will tell you how much memory you have malloced and not freed

Malloc and Free

`malloc` and `free` are a major source of bugs in C programs

Malloc and Free

`malloc` and `free` are a major source of bugs in C programs

- Using memory you have not `malloc`ed

Malloc and Free

`malloc` and `free` are a major source of bugs in C programs

- Using memory you have not `malloc`ed
- `free`ing memory more than once

Malloc and Free

`malloc` and `free` are a major source of bugs in C programs

- Using memory you have not `malloc`ed
- `free`ing memory more than once
- Using memory already `free`d

Malloc and Free

`malloc` and `free` are a major source of bugs in C programs

- Using memory you have not `malloc`ed
- `free`ing memory more than once
- Using memory already `free`d
- Accessing beyond the ends of the allocated space

Malloc and Free

`malloc` and `free` are a major source of bugs in C programs

- Using memory you have not `malloc`ed
- `free`ing memory more than once
- Using memory already `free`d
- Accessing beyond the ends of the allocated space
- Overwriting pointers, creating garbage

Malloc and Free

`malloc` and `free` are a major source of bugs in C programs

- Using memory you have not `malloc`ed
- `free`ing memory more than once
- Using memory already `free`d
- Accessing beyond the ends of the allocated space
- Overwriting pointers, creating garbage
- And so on

Malloc and Free

On the other hand, `malloc` and `free` are extremely useful in the right hands

Malloc and Free

On the other hand, `malloc` and `free` are extremely useful in the right hands

- the programmer has precise control on the allocation of memory

Malloc and Free

On the other hand, `malloc` and `free` are extremely useful in the right hands

- the programmer has precise control on the allocation of memory
- they concentrate the programmer's attention towards the efficient use of memory

Malloc and Free

On the other hand, `malloc` and `free` are extremely useful in the right hands

- the programmer has precise control on the allocation of memory
- they concentrate the programmer's attention towards the efficient use of memory
- they are reasonably fast

Malloc and Free

On the other hand, `malloc` and `free` are extremely useful in the right hands

- the programmer has precise control on the allocation of memory
- they concentrate the programmer's attention towards the efficient use of memory
- they are reasonably fast
- the programmer can tune their use to the problem in hand

Malloc and Free

Exercise. What is the bug here?

```
int a[10];
...
free(a);
```

Exercise. `malloc` and `free` are fast, but not free: they take some time (and some overhead space) to manage memory. Find out how much of an overhead they incur on your computer

Exercise. Compare this with Java's memory management

Exercise. Look up `alloca` and dynamic stack allocation

Malloc and Free

Exercise. Deliberately write bad code that does these kinds of things. Run it and see what goes wrong. Use `valgrind` on your code

Exercise. Deliberately write good code that avoids these kinds of things

Malloc and Free

Exercise. Think about the symmetry:

```
int *a = malloc(...);
free(a);
```

giving a pointer that points at a non-object; and

```
int *a = malloc(...);
a = malloc(...);
```

giving an object that no pointer pointing to it

Malloc and Free

Having `malloc` and `free` is simultaneously one of the great strengths of C and one of its great weaknesses

Malloc and Free

Having `malloc` and `free` is simultaneously one of the great strengths of C and one of its great weaknesses

Some languages, for example, Java, have *automatic memory management*

Malloc and Free

Having `malloc` and `free` is simultaneously one of the great strengths of C and one of its great weaknesses

Some languages, for example, Java, have *automatic memory management*

This is when the system manages the memory for the programmer so they don't have to allocate and free objects themselves

Malloc and Free

Having `malloc` and `free` is simultaneously one of the great strengths of C and one of its great weaknesses

Some languages, for example, Java, have *automatic memory management*

This is when the system manages the memory for the programmer so they don't have to allocate and free objects themselves

Java's `new` is like `malloc`. There is no analogue to `free`

Malloc and Free

Having `malloc` and `free` is simultaneously one of the great strengths of C and one of its great weaknesses

Some languages, for example, Java, have *automatic memory management*

This is when the system manages the memory for the programmer so they don't have to allocate and free objects themselves

Java's `new` is like `malloc`. There is no analogue to `free`

Java programs generate garbage at a prodigious rate

Malloc and Free

So the Java system has to clear up the garbage itself, else it too would run out of memory

Malloc and Free

So the Java system has to clear up the garbage itself, else it too would run out of memory

So Java includes (as part of the Java system) a *garbage collector* that periodically trawls through memory looking for inaccessible garbage: chunks of memory that can never be accessed in the program as the program has overwritten/lost the pointers to those chunks

Malloc and Free

So the Java system has to clear up the garbage itself, else it too would run out of memory

So Java includes (as part of the Java system) a *garbage collector* that periodically trawls through memory looking for inaccessible garbage: chunks of memory that can never be accessed in the program as the program has overwritten/lost the pointers to those chunks

It collects the areas of garbage memory together and then can allocate those bytes in subsequent calls

Malloc and Free

So the Java system has to clear up the garbage itself, else it too would run out of memory

So Java includes (as part of the Java system) a *garbage collector* that periodically trawls through memory looking for inaccessible garbage: chunks of memory that can never be accessed in the program as the program has overwritten/lost the pointers to those chunks

It collects the areas of garbage memory together and then can allocate those bytes in subsequent calls

Note it is safe to reallocate those bytes as by definition garbage is inaccessible to the program, thus reusing them can have no effect on the program

Malloc and Free

This seems wonderful, so why does C use these problematic `malloc` and `free`?

Malloc and Free

This seems wonderful, so why does C use these problematic `malloc` and `free`?

It a choice of trade-offs

Malloc and Free

This seems wonderful, so why does C use these problematic `malloc` and `free`?

It a choice of trade-offs

Automatic memory management:

Malloc and Free

This seems wonderful, so why does C use these problematic `malloc` and `free`?

It a choice of trade-offs

Automatic memory management:

- Releases the programmer from having to worry about memory

Malloc and Free

This seems wonderful, so why does C use these problematic `malloc` and `free`?

It a choice of trade-offs

Automatic memory management:

- Releases the programmer from having to worry about memory
- Should never go wrong

Malloc and Free

This seems wonderful, so why does C use these problematic `malloc` and `free`?

It a choice of trade-offs

Automatic memory management:

- Releases the programmer from having to worry about memory
- Should never go wrong
- Encourages sloppy programming

Malloc and Free

This seems wonderful, so why does C use these problematic `malloc` and `free`?

It a choice of trade-offs

Automatic memory management:

- Releases the programmer from having to worry about memory
- Should never go wrong
- Encourages sloppy programming
- Has a significant time and space overhead in management and garbage collection

Malloc and Free

Manual memory management:

Malloc and Free

Manual memory management:

- Requires the programmer to think carefully about memory usage

Malloc and Free

Manual memory management:

- Requires the programmer to think carefully about memory usage
- Encourages careful use of memory

Malloc and Free

Manual memory management:

- Requires the programmer to think carefully about memory usage
- Encourages careful use of memory
- Can be tuned for a specific application

Malloc and Free

Manual memory management:

- Requires the programmer to think carefully about memory usage
- Encourages careful use of memory
- Can be tuned for a specific application
- Is a frequent source of errors

Malloc and Free

It's your choice and should be taken into account when you are choosing a programming language to implement a project

Malloc and Free

It's your choice and should be taken into account when you are choosing a programming language to implement a project

C does have bolt-on garbage collectors, if you really want them

Malloc and Free

What do mean when we say `malloc` “allocates some bytes”?

Malloc and Free

What do mean when we say `malloc` “allocates some bytes”?

It means a reservation is made on a chunk of bytes from the program’s memory

Malloc and Free

What do mean when we say `malloc` “allocates some bytes”?

It means a reservation is made on a chunk of bytes from the program’s memory

The reservation exists until we do a `free`

Malloc and Free

This means this kind of code is OK (and very common):

```
struct intlist *make(int v)
{
  struct intlist *newl;
  ...
  newl = (struct intlist *)malloc(sizeof(struct intlist));
  ...
  return newl;
}
```

as the reservation persists beyond the end of the function call `make`, so the returned pointer remains valid outside of the function call

Malloc and Free

However, this is bad:

```
struct intlist *make(int v)
{
  struct intlist newl;
  ...
  return &newl;
}
```

as the structure `newl` will only exist for the duration of the function call

Malloc and Free

However, this is bad:

```
struct intlist *make(int v)
{
  struct intlist newl;
  ...
  return &newl;
}
```

as the structure `newl` will only exist for the duration of the function call

By “exist” we mean “is valid”. It’s still there in memory!

Malloc and Free

Because the pointer returned is still a pointer to somewhere in memory, the code might even work for a while

Malloc and Free

Because the pointer returned is still a pointer to somewhere in memory, the code might even work for a while

Until you have another function call that extends the stack again to cover that place where your structure lives. And then overwrites it with whatever locals and arguments that function requires

Malloc and Free

Moral: don’t return pointers to things on the stack

Malloc and Free

Moral: don't return pointers to things on the stack

More precisely: it is OK to use pointers to things on the stack *while that frame is still active*. Thus using such a pointer within the current function is fine; as is passing the pointer down to "deeper" functions. But you must never return the pointer up to a place where the frame has gone

Malloc and Free

Moral: don't return pointers to things on the stack

More precisely: it is OK to use pointers to things on the stack *while that frame is still active*. Thus using such a pointer within the current function is fine; as is passing the pointer down to "deeper" functions. But you must never return the pointer up to a place where the frame has gone

Exercise. Investigate to see what happens when you return pointers to things on the stack

Malloc and Free

Exercise. What about

```
struct intlist make(int v)
{
  struct intlist newl;
  ...
  return newl;
}
```

Files

You will need to manipulate files: read and write data

Files

You will need to manipulate files: read and write data

C provides two principal kinds of access to files:

- unbuffered
- buffered

Files

You will need to manipulate files: read and write data

C provides two principal kinds of access to files:

- unbuffered
- buffered

We shall look at buffered I/O: it's the one you will use the most

Files

You will need to manipulate files: read and write data

C provides two principal kinds of access to files:

- unbuffered
- buffered

We shall look at buffered I/O: it's the one you will use the most

If you need unbuffered I/O, you will easily be able to pick it up for yourself

Files

The major operations on files are

- open, close (functions `fopen`, `fclose`)
- read, write (functions `fread`, `fwrite`, `fprintf`)

Files

Buffered I/O in C uses a pre-defined structure, a `FILE`

Files

Buffered I/O in C uses a pre-defined structure, a `FILE`

The internal details of this structure are unimportant, and it will all be pre-declared for you as long as you `#include <stdio.h>`

Files

Buffered I/O in C uses a pre-defined structure, a `FILE`

The internal details of this structure are unimportant, and it will all be pre-declared for you as long as you `#include <stdio.h>`

In fact, you will always be using a pointer to a `FILE`, a `FILE*`

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  FILE *in, *out;
  char buf[1024]; // chunk of bytes
  int nread;

  in = fopen(argv[1], "r");  // ought to check
  out = fopen(argv[2], "w"); // for success

  do {
    nread = fread(buf, 1, 1024, in);
    fwrite(buf, 1, nread, out);
  } while (nread > 0);

  fclose(in);
  fclose(out);
  return 0;
}
```

Files

- We declare variables `in` and `out` of type `FILE*`

Files

- We declare variables `in` and `out` of type `FILE*`
- `fopen` opens a file for reading or writing or both; the argument `"r"` or `"w"` tells it which (also: `r+` for read+write, `a` for append and several others)

Files

- We declare variables `in` and `out` of type `FILE*`
- `fopen` opens a file for reading or writing or both; the argument `"r"` or `"w"` tells it which (also: `r+` for read+write, `a` for append and several others)
- `fopen` returns a pointer to a `FILE` object that we shall use to refer to the files (and it does some `malloc`ing behind the scenes to allocate that structure)

Files

- We declare variables `in` and `out` of type `FILE*`
- `fopen` opens a file for reading or writing or both; the argument `"r"` or `"w"` tells it which (also: `r+` for read+write, `a` for append and several others)
- `fopen` returns a pointer to a `FILE` object that we shall use to refer to the files (and it does some `malloc`ing behind the scenes to allocate that structure)
- We should check for success of both `fopen`s. They will return `NULL` if they failed. For example, trying to read a file that does not exist or we do not have permission to read

Files

```
nread = fread(buf, 1, 1024, in);
fwrite(buf, 1, nread, out);
```

- We repeatedly read bytes from `in`. We shall try to read 1024 items of size 1 byte each into the buffer `buf`

Files

```
nread = fread(buf, 1, 1024, in);
fwrite(buf, 1, nread, out);
```

- We repeatedly read bytes from `in`. We shall try to read 1024 items of size 1 byte each into the buffer `buf`
- In comparison, to read `n` integers we could have written `fread(intbuf, sizeof(int), n, in);` where `intbuf` is a pointer to an area of memory (array) big enough to hold `n` integers.

Files

```
nread = fread(buf, 1, 1024, in);
fwrite(buf, 1, nread, out);
```

- We repeatedly read bytes from `in`. We shall try to read 1024 items of size 1 byte each into the buffer `buf`
- In comparison, to read `n` integers we could have written `fread(intbuf, sizeof(int), n, in);` where `intbuf` is a pointer to an area of memory (array) big enough to hold `n` integers.
- `fread` returns the number of *items* actually read, the number of bytes in our example; the number of `int`s in the above

Files

```
nread = fread(buf, 1, 1024, in);
fwrite(buf, 1, nread, out);
```

- We repeatedly read bytes from `in`. We shall try to read 1024 items of size 1 byte each into the buffer `buf`
- In comparison, to read `n` integers we could have written `fread(intbuf, sizeof(int), n, in);` where `intbuf` is a pointer to an area of memory (array) big enough to hold `n` integers.
- `fread` returns the number of *items* actually read, the number of bytes in our example; the number of `int`s in the above
- We write that number of bytes to `out`

Files

```
nread = fread(buf, 1, 1024, in);
fwrite(buf, 1, nread, out);
```

- We repeatedly read bytes from `in`. We shall try to read 1024 items of size 1 byte each into the buffer `buf`
- In comparison, to read `n` integers we could have written `fread(intbuf, sizeof(int), n, in);` where `intbuf` is a pointer to an area of memory (array) big enough to hold `n` integers.
- `fread` returns the number of *items* actually read, the number of bytes in our example; the number of `int`s in the above
- We write that number of bytes to `out`
- We repeat until there are no more bytes to read

Files

- A careful programmer would check the return from `fwrite` to ensure all the data was successfully written (e.g., disk full)

Files

- A careful programmer would check the return from `fwrite` to ensure all the data was successfully written (e.g., disk full)
- We then close `in` and `out`

Files

- A careful programmer would check the return from `fwrite` to ensure all the data was successfully written (e.g., disk full)
- We then close `in` and `out`
- It is important to close files, particularly when writing, to ensure all the data is safely written to disk before the program ends

Files

- A careful programmer would check the return from `fwrite` to ensure all the data was successfully written (e.g., disk full)
- We then close `in` and `out`
- It is important to close files, particularly when writing, to ensure all the data is safely written to disk before the program ends
- Also, `fclose` does a `free` of the relevant datastructures that `fopen` made behind the scenes

Files

When your program starts, the system supplies three pre-opened `FILE*`s for your convenience

Files

When your program starts, the system supplies three pre-opened `FILE*`s for your convenience

- `stdin` opened to read from the keyboard

Files

When your program starts, the system supplies three pre-opened `FILE*`s for your convenience

- `stdin` opened to read from the keyboard
- `stdout` opened to write to the screen

Files

When your program starts, the system supplies three pre-opened `FILE*`s for your convenience

- `stdin` opened to read from the keyboard
- `stdout` opened to write to the screen
- `stderr` opened to write to the screen

Files

When your program starts, the system supplies three pre-opened `FILE*`s for your convenience

- `stdin` opened to read from the keyboard
- `stdout` opened to write to the screen
- `stderr` opened to write to the screen

It is useful to have two ways of standard output: one for normal output and one for error output

Files

When your program starts, the system supplies three pre-opened `FILE*`s for your convenience

- `stdin` opened to read from the keyboard
- `stdout` opened to write to the screen
- `stderr` opened to write to the screen

It is useful to have two ways of standard output: one for normal output and one for error output

Using command-line shells we can redirect the two kinds of output to different places

Files

```
fwrite(str, 1, 12, stdout);
```

is an unlikely way of writing a string to the screen

Files

`fwrite(str, 1, 12, stdout);`
is an unlikely way of writing a string to the screen

Exercise. Look at the `man` pages for these file functions, particularly `fopen`

Files

Another useful function is `fprintf`

Files

Another useful function is `fprintf`

This is just like `printf`, but outputs to a `FILE*` rather than the screen

Files

Another useful function is `fprintf`

This is just like `printf`, but outputs to a `FILE*` rather than the screen

In fact, `printf` is the same as `fprintf(stdout, ...)`

Files

Another useful function is `fprintf`

This is just like `printf`, but outputs to a `FILE*` rather than the screen

In fact, `printf` is the same as `fprintf(stdout, ...)`

And `fprintf(stderr, ...)` is the way you usually report errors to the user

Files

Exercise. Look at `fscanf` (and `scanf`), the "opposite" to `printf` that reads text formatted input

Exercise. Make sure you understand the distinction between using `fread` to read a (4 byte, say) integer and using `fscanf` to read a (character string) integer

Exercise. Look up `feof`, `fflush` and `ferror`

Exercise. Read `man stdio`