

# CITS3007 Secure Coding

## C language, intro to buffer overflows

Unit coordinator: Arran Stewart

# Outline

- ▶ C language topics – parts of you C should know
- ▶ Systems programming refresher – privilege levels and system calls
- ▶ Vulnerabilities – buffer overflows

## C language refresher

# Why C?

Why do we use C in this unit, instead of some other language (Python, or C#, or Rust, say)?

# Why C?

C has a privileged place in the software ecosystem.

- ▶ Most modern operating systems (e.g. Linux, Windows and macOS) are written in C
  - ▶ their *interfaces* are defined in C
- ▶ Many programming languages have their primary implementation in C (e.g. Python, JavaScript, Lua, Bash)

So C underpins many modern systems and languages.

# Why C?

- ▶ C often serves as a “lingua franca” when extending languages or developing programs written in multiple languages
- ▶ For instance, the Python language can be extended by writing **new built-in modules** in C.

## Features

C was created as an **efficient systems programming language**, and was first used to re-write portions of the Unix operating system so as to make them more portable.

It aims to give the programmer a **high level of control** over the organization of data and the operations performed on that data.

But it also assumes the programmer **knows what they are doing**, and provides very little in the way of safeguards.

# Features

C inherited some features from the language [PL/I](#), but unfortunately in some cases opted for less security than PL/I.

For instance, [buffer overflows](#) (which we look at shortly) were rare in PL/I, as it required that programmers always specify a maximum length for strings:<sup>1</sup> C does not implement this feature.

---

<sup>1</sup>Karger & Schell (2002)



# Features

C leaves many details about the behaviour of programs (for instance, what range of numbers an `int` can hold) to the compiler, and the details can vary from platform to platform.

The intention is to allow the compiler to use the **most efficient types** and **most efficient processor instructions** for the platform it is targeting.

# Language standards

We will largely discuss the C11 standard,<sup>2</sup> which is still in widespread use.

That said, as long as your code compiles and runs correctly using the standard CITS3007 development environment, you are welcome to use later versions of the language if you wish.<sup>3</sup>

---

<sup>2</sup>ISO/IEC 9899:2011. See [ISO/IEC 9899:201x](https://www.open-std.org) at <https://www.open-std.org> for a draft version.

<sup>3</sup>gcc can be instructed to use C17, for instance, by passing `-std=c17` to the compiler.

# Language references and texts

If you're not already familiar with C, you will need to get up to speed in the first few weeks through self-study.

See the [website](#) for textbook recommendations.

Robert Seacord has a textbook which I quite like, but you should pick a textbook that you feel comfortable with.

## Language references and texts

If you are already familiar with C:

- ▶ The **ISO/IEC C11 standard** is a bit wordy, and the vocabulary takes a bit of getting used to – but it's not *that* difficult to follow, and it's the final word on what a legal C11 program should do.
- ▶ **<https://cppreference.com>** actually has very good coverage of C header files and functions.  
Just make sure you're reading the right one.

- ▶ From a corresponding C++ page, follow the “C language” links down the bottom of page
- ▶ C language topics should have a URL that looks like  
<https://en.cppreference.com/w/c/SOMETHING>

# These slides != a textbook

Please note: these lecture slides aim to refresh your memory on details of the C language, and highlight some important differences from other languages.

They are not a complete reference, nor are they a substitute for a C textbook.

If you rely on them to explain all the details of the C language, you will probably get questions wrong in the assessments, and then you will be unhappy.

# Major surprises

Some of the following features of C often surprise people coming from other languages:

- ▶ (Almost) everything is an integer (or derived from an integer type)
- ▶ Assignment (“=”) will only sometimes do what you think it should do
- ▶ If you misuse memory (e.g. going outside the bounds of an array), you get no warnings or exceptions about this – the compiler assumes you know what you’re doing
  - ▶ Instead of exceptions, the behaviour of your program becomes **undefined** – it literally has no meaning, is not a valid C program, and the compiler is allowed to generate whatever compiled code it likes.
- ▶ Many aspects of program behaviour are not fixed by the language standard, but are **implementation-defined**.

# Definedness

The C language standard gives great latitude to compiler implementers to do whatever is most efficient for a particular platform.

Instead of specifying exactly what some construct will do, the standard leaves this up to the compiler implementer.

# Definedness

There are three different types of “not-precisely-defined-by-the-standard” behaviour:

- implementation-defined** The implementation must pick some behavior, and its choice must be documented.  
(Query: What does “implementation” mean? A compiler? A version of that compiler? A version of that compiler, targeting a specific platform?)
- unspecified** Similar to implementation-defined, except that the choice need not be documented.  
(The choice need not be deterministic or consistent – an implementation could choose different behaviours at different times.)
- undefined** Anything at all can happen; the standard imposes no requirements. The program might fail to compile, or it might execute incorrectly, or it might by pure luck do exactly what the programmer intended.

(See <https://c-faq.com/ansi/undef.html> for more details.)



## Definedness exercise

Suppose you have a function `eraseAll` in a C program, left behind by some other programmer:

```
static int eraseAll() {  
    return system("rm -rf /");  
}
```

Running this function would invoke the `rm` command, and request it to delete all files on your system (thus destroying the system).

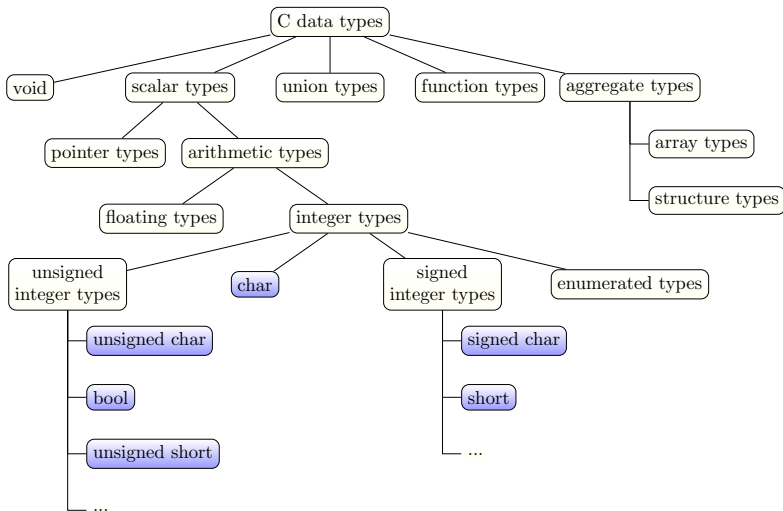
Fortunately, however, you never actually call the `eraseAll` function from your code.

## Definedness exercise

Suppose that somewhere in the code your program *does* invoke, you accidentally cause undefined behaviour (for instance, by going out of bounds of an array, or trying to dereference an uninitialized pointer).

Can the compiler output a program which calls the `eraseAll` function and destroys your system?

# C data types



See `<stdint.h>` for many more integer types (e.g. fixed-width integer types like `int32_t`).

# “Imaginary” types

*Before you can understand strings in C, you have to realize the truth. C has no strings.*

— /u/Different-Brain-9210 on [Reddit](#)

## “Imaginary” types

Unlike many other languages, C does **not have** a “string” type.

There is no type in C called “string”, representing human-readable text.

What C has instead are **arrays of chars**, some of which might represent strings, and some of which might not.

## “Imaginary” types

You can think of C as having two “imaginary” types, which it’s up to the programmer to keep track of in their head:

“blob of bytes”

Raw access to a sequence of contiguous bytes in memory. Other languages sometimes call this type `bytes` or a `bytestring`.<sup>4</sup>

“string”

A human-readable string of text. To work properly with string-related library functions, they must be terminated with a null character (usually written `'\0'`).

If the terminator is missing, that will result in security problems.

Both these “imaginary” types are represented in code as arrays of `chars`.

<sup>4</sup>You can also think of `char *` as sometime being a type called “view-this-row-memory-as-a-bytestring”. It’s an exception to the normal rule that you must never access a location in memory by the “wrong” type.

# Array decay

C semantics is based on there being **arrays** sitting in memory at various locations.

All arrays always have an exact size, and if you go outside the bounds of the array, that will result in security problems.

Unfortunately, unless you are careful, it's easy for the information about array length to “vanish” from the programmer's view – more on this in labs.

# Integers in C

C has a large number of integral data types.<sup>5</sup> The most common are:

## standard integer types

- ▶ standard signed integer types: `signed char`, `short int`, `int`, `long int`, and `long long int`
- ▶ standard unsigned integer types: `_Bool` (also available as `bool`), `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`, and `unsigned long long int`
- ▶ the `char` type.

What range of integers these can hold, and which of these types are equivalent to each other, is implementation dependent.

---

<sup>5</sup>In fact, nearly every type you see in this unit (besides function types) is either an integer type, or derived from (array or struct or pointer to) integer types.



# Integers in C

## standard integer types

- ▶ standard signed integer types: `signed char`, `short int`, `int`, `long int`, and `long long int`
- ▶ standard unsigned integer types: `_Bool` (also available as `bool`), `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`, and `unsigned long long int`
- ▶ the `char` type.

The C11 standard states that the `char` type is equivalent to *either* `signed char` or `unsigned char`, but which one is the case is implementation-defined.

It says the size of a `char` is one byte, and has at least 8 bits, but *doesn't* otherwise constrain how many bits exist in a byte (this, too, is implementation-defined).<sup>6</sup>

---

<sup>6</sup>The `CHAR_BIT` macro in `<limits.h>` will tell you the number of bits per byte. On Unix-like OSs, the macro `NBBY`, defined in `<sys/param.h>`, will give the same result.

## Floating point types

C also has three “real floating types”, but we will be less concerned with them.

## real floating types

- ▶ float
- ▶ double
- ▶ long double

(It also has three corresponding types for **complex numbers**, which we won't use at all.)

# Functions in C

All executable statements in C must be written inside a procedure – C calls its procedures “functions”.

C functions may return a result, in which case the signature of the function will indicate the return type. For instance:

```
int square(int x);
```

The function declared above takes one argument (an **int**), and returns an **int** value.

# Functions in C

```
void print_int_to_terminal(int x);
```

Alternatively, a function may be declared as having return type **void**, in which case it *doesn't* return any value as a result.

Both void and non-void functions may have *side effects*: they may for instance modify the values of global variables, perform output to the terminal, or alter the state of files or attached devices.

# Function declarations and definitions

A function **declaration** “tells” code following it about a function:

```
int square(int x);
```

A function **definition** provides the “body” of the function:

```
int square(int x) {  
    return x * x;  
}
```

# Function conventions in C

There are two types of functions in C:

- ▶ Functions that can *fail* – they try to do something, but may sometimes not succeed, even when called correctly.

Examples: `fopen`, `malloc`.

- ▶ Functions that cannot fail – if called correctly, these always succeed.

Examples: `strlen`, `memcpy`, `isalpha`.

However, it can be easy to call both these sorts of functions incorrectly.

# Function conventions in C

The convention in C for function return values is as follows:

## Functions that can fail

If the function normally returns a pointer – it will return `NULL` to indicate failure.

If the function normally returns a non-negative `int` – it will return -1 to indicate failure.

More on these in the labs.

# Control structures

C has the following control flow structures:

selection statements

if and switch statements

loops

while, do and for loops

jumps

continue, break, goto and return statements



# Control structures

C has the following control flow structures:

selection statements

## if and switch statements

loops

## while, do and for loops

jumps

## continue, break, goto and return statements

The `goto` statement *is* useful in C. One reason is that C does not have exceptions and “`finally`” blocks (which can be used to handle errors and execute “clean-up” code in Java and Python).

`goto` can be used to jump to an error-handling section of your function (see your C textbook, or Seacord chap. 5, for details).

# Scope in C

C has two basic types of scope:

- ▶ **global scope** (or “file scope”): for variables declared outside all functions. These are visible from the declaration, to the end of the file.
- ▶ **block scope**: for variables declared within a function or statement block. These are visible from the declaration, to the end of the function or statement block.

For global variables (and for functions, which are always global – C doesn't have nested functions): adding the keyword **static** before them ensures that the variable or function is *only* visible from within that file.

Limiting scope is C's primary method of implementing **information hiding**.

# Scope in C

global, usable  
in any file

from  
another file

global, this  
file only

```
int OUR_NUM = 42;
int OTHER_NUM ;
static int OUR_PRIVATE_NUM ;
int multiply (int m, int n) {
    int i, res = 0;
    for (i = 0; i < n; i++) {
        int tmp = res + m;
        res = tmp;
    }
    return res;
}
```

local variable  
declarations

local to block

# Arrays in C

C provides support for 1-dimensional and multi-dimensional arrays.

## 1-dimensional array

```
#define ARRAY_SIZE 10
int some_array[ARRAY_SIZE];
```

## 2-dimensional array

```
#define ARRAY_HEIGHT 5
#define ARRAY_WIDTH 10
int two_d_array[ARRAY_HEIGHT][ARRAY_WIDTH];
```

# Size of arrays

For our purposes – *the size of arrays in C is fixed at compile-time.*

You shouldn't declare an array using a variable as a size, even if the compiler lets you.

```
int mylen = 10;
char myarray[mylen];
```

Using such “variable-length arrays” is error-prone in C, and can often allow an attacker to exploit [stack-clashing vulnerabilities](#).

# Strings in C

C does not provide a separate datatype for strings – rather, strings are considered to be arrays of **chars**, with the **NUL** character (which has ASCII code 0) acting as a terminator.

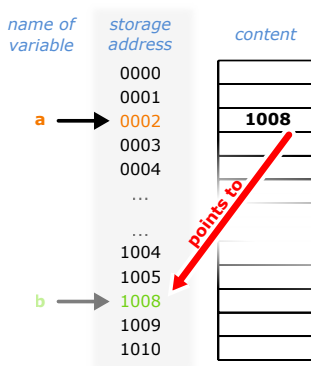
## String

```
// this declaration:  
char my_str[] = "cat";  
// is equivalent to:  
char my_str[4] = { 'c', 'a', 't', '\0'};
```

# Pointers in C

**Pointer** types in C hold a reference to an entity of some *other* type. For instance a “pointer to `int`” (written `int *`) holds a reference to an `int`.

It's usually convenient to think of this “reference” as the address of a location in memory, but the C11 standard does not require that to be the case.



A pointer and the variable it references<sup>1</sup>

<sup>1</sup>Image courtesy of Wikipedia,

# Pointers in C

C allows the use of **pointer arithmetic**. In addition to performing (say) addition on two integer values, we can perform it on one pointer value and one integer value.

```
int * p1 = NULL;
int * p2 = p1 + 4;
```

Adding 4 to a pointer doesn't move it along by 4 bytes. (What does it do?)

We can also subtract one pointer from another, and perform equality and inequality comparisons on two pointers (`==`, `<`, `>`, `<=`, and `>=`).



# Pointers in C

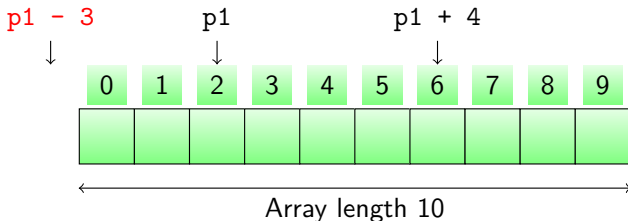
Many languages disallow pointer arithmetic, since its use can easily result in invalid pointers (pointers that do not reference a properly initialized object of the correct type).

C allows it; it is up to the programmer to ensure they comply with the standard's rules as to when a pointer is valid.

If the programmer fails to comply with those rules, the result usually is that the behaviour of the program is *undefined*.

(In other words: the program has no well-defined “meaning”, according to the C11 standard; and the standard places no constraints on what behaviour it may have.)

# Pointers in C



For instance: if arithmetic is performed on a pointer which references some element of an array, and the resulting pointer would go outside the bounds of the array,<sup>7</sup> then the behaviour of the program is undefined.

<sup>7</sup>To be precise: the pointer must point either to an element of the array, or the position one past the last element.

# Pointers in C

A pointer to a variable can be obtained using the `'&'` (“address-of”) operator, and pointers can be dereferenced using `'*'` (the dereference operator).

```
int some_num = 42;
int * num_addr = &some_num;
*num_addr = 99;
printf("the number is: %d\n", some_num);
// prints "the number is 99"
```

# Lifetime

Variables have a **storage duration** that determines their “lifetime”.

- ▶ Memory for **global** variables is allocated when the program starts running, and persists until the program exits
- ▶ However, the majority of variables in a program are **local** variables, and have what is called “automatic storage duration”
  - ▶ This basically means they “disappear” when the function they are declared in exits, and the memory allocated to them is reclaimed
  - ▶ If you’ve somehow managed to hang onto a reference to this memory, the behaviour of your program is *undefined*

## Automatic lifetime and dangling pointers

Consider this function:

```
int * myfunc() {
    int a_local_var = 36;
    int * a_pointer = &a_local_var;
    return a_pointer;
}
```

There's nothing wrong with returning a pointer – lots of functions do it (like the standard function `getenv` – `char* getenv (const char* name)` – which gives you the value of an environment variable).

But a caller of `myfunc` will receive a pointer to memory which has been reclaimed – a “**dangling pointer**” – and such a pointer results in undefined behaviour.

## dangling pointers

myfile.c

```
int * myfunc() {
    int a_local_var = 36;
    int * a_pointer = &a_local_var;
    return a_pointer;
}
```

← dangling pointer

Compilers will generally not warn you about this – the above code compiles with `gcc -pedantic -Wall -Wextra` with no warnings.

Code **static analyzers** exist which *will* warn you – more about them, later.

e.g. `clang-tidy myfile.c` will give the output

```
1 warning generated.
myfile.c:4:3: warning: Address of stack memory associated with local
variable 'a_local_var' returned to caller [clang-analyzer-core.StackAddrEscapeBase]
    return a_pointer;
           ^
```

# Dynamically allocated memory

- ▶ Data which we want to persist beyond the execution time of a function needs either to be global, or to be allocated in a region of memory called the **heap**.
- ▶ Memory allocated on the heap is said to be “**dynamically allocated**”
- ▶ The primary C functions used to manage dynamic memory are
  - ▶ **malloc**, for allocating memory, and
  - ▶ **free**, for releasing it.

```
void *malloc(size_t size);  
void free(void *ptr);
```

## Dynamically allocated memory

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

int* make_arr(int n) {
    int* arr = malloc(n * sizeof(int));
    return arr;
}

int main() {
    int n;
    printf("How big an array to allocate? ");
    scanf("%d",&n); // usually, prefer
                     // string parsing functions
                     // like strtol
    int* arr = make_arr(n);
    for(i = 0; i < n; i++)
        arr[i] = n;
    free(arr);
}
```



## Dynamically allocated memory

- ▶ Once a pointer has been freed, using that pointer's value at all – even without dereferencing it – is undefined behaviour.

```
int *p = malloc(sizeof(int));
free(p);
if (p == NULL) {
    // ...
}
```

- ▶ So is calling `free` on a pointer more than once.
- ▶ Attempting to read from `malloced` memory before it has been initialized results in an “indeterminate value” – not undefined, but almost certainly not what you want

## Memory: call stack

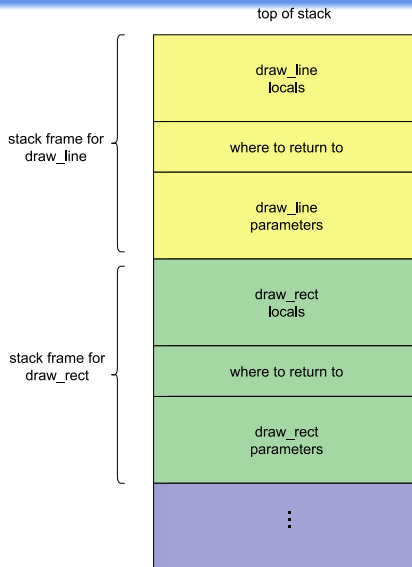
On most architectures, calls to C functions work something like this:

- ▶ Every time a C function starts executing, space is allocated for its parameters and local variables on the **call stack**
  - ▶ for each function that is entered, a **stack frame** gets pushed *onto* the call stack
  - ▶ the stack frame consists of enough memory to store the function parameters, local variables and a record of where to return to
  - ▶ when the function is exited, a stack frame gets taken *off* the call stack

## Memory: call stack

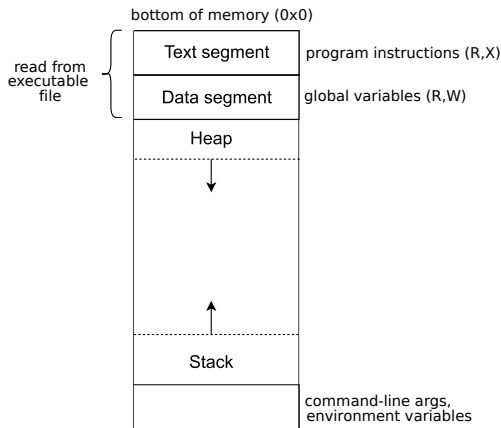
```
void draw_line(point* p1, point* p2){
    // ...
}

void draw_rect(point* topLeft, point* botRight){
    point p1 = {.x=topLeft->x, .y=topLeft->y };
    point p2 = {.x=botRight->x, .y=topLeft->y };
    draw_line(&p1, &p2);
    // ...
}
```



## Memory: process memory layout

The layout of a process's data in virtual memory looks something like this.



On Linux, `cat /proc/some_pid/maps` shows the virtual address space of a process. (Try `cat /proc/self/maps` to get the address space of the `cat` process itself.)

The **text** segment is typically made *shareable*, so that multiple processes can be run from one executable file and share a single copy (safe, since it's read-only).

## Typedefs

C allows types to be given “aliases”, using the `typedef` keyword. The original type comes *first*, then the alias.

```
typedef int colour;
```

# Structs

C provides **structs** to create composite data types (“product types”) in which a related set of variables can be grouped together in one contiguous block of memory.

```
struct address {  
    char * street_number;  
    char * street_name;  
    char * suburb;  
    int postcode;  
};  
  
void my_func() {  
    // we can initialize ...  
    struct address some_addr = { // like this:  
        "13a", "Cooper St", "Nedlands", 6009  
    };  
    struct address other_addr = { // or like this (since C99)  
        .postcode = 6009, .suburb = "Nedlands",  
        .street_number = "13a", .street_name = "Cooper St"  
    };  
}
```

## Struct members

```
typedef struct {
    char * street_number;
    char * street_name;
    char * suburb;
    int postcode;
} address;
```

struct members can be accessed using the "." (member access) operator.

If, rather than a struct, you have a *pointer* to a struct, use the “->” (member access through pointer) operator.

```
void my_func(struct address a, struct address *pa) {
    printf("postcode of a: %d\n", a.postcode);
    printf("postcode of pa: %d\n", pa->postcode);
}
```

## enums

C allows user-defined data types which assign meaningful names to integral constants:

```
enum shape_operation {
    draw = -1,
    move,
    delete = 4,
    hide
};
```

Enumerated types are *integer types*, and so can be used anywhere an integer could be. As a result, they offer no real *type safety*: nothing distinguishes an `enum shape_operation` from (say) a `signed int`.<sup>8</sup>

<sup>8</sup>Each enumerated type is compatible with some *integral* type which can hold all the values, but it's implementation-defined what type that is.



# Unions

A C **union** may hold *multiple* different types, of different sizes – but only one type at a time.

For instance, suppose we receive a “blob” of data from over the network which represents a message. The first 8 bits (1 byte) are a code that tell us what the rest of the “blob” means:

- ▶ 0 indicates it's a double
- ▶ 1 indicates it's an int

# Unions

We could use the following to represent these messages:

```
union double_or_int {  
    double d;  
    int i;  
};  
  
struct message {  
    char message_type;  
    union double_or_int;  
};
```

# Unions

```
union double_or_int {
    double d;
    int i;
};

struct message {
    char message_type;
    union double_or_int;
};
```

We can then correctly decode a message with code like this:

```
void decode_message(struct message * m) {
    if (m.message_type == 0) {
        double d = m->d;
        printf("It's a double: %f\n", d);
    } else if (m.message_type == 1) {
        int i = m->i;
        printf("It's an int: %d\n", i);
    }
}
```

# Unions – a problem

```
void decode_message(struct message * m) {  
    if (m.message_type == 0) {  
        double d = m->d;  
        printf("It's a double: %f\n", d);  
    } else if (m.message_type == 1) {  
        int i = m->i;  
        printf("It's an int: %d\n", i);  
    }  
}
```

We've *assumed* here that a **char** is 8 bits in size. And on every reasonable platform available today, it is (but see [here](#)).

If we want to make sure, we can use C11's **static assert** feature to verify the size.

```
#include <assert.h>  
#include <limits.h>  
// This will be checked at compile time.  
static_assert(CHAR_BIT == 8, "only works if a char is 8 bits");
```

# Function pointers

Pointers to *functions* can be passed around and used in C.

The syntax for function pointers is not especially pleasant.

```
// pointer to a void function taking an int
```

```
void (*func_ptr)(int);
```

```
void use_ptr(void (*p)(int)) {
```

```
    p(42); // call pointed-to function
```

```
}
```

```
void print_num(int n) {
```

```
    printf("the number is %d\n", n);
```

```
}
```

```
int main() {
```

```
    func_ptr = print_num;
```

```
    use_ptr(func_ptr);
```

```
}
```

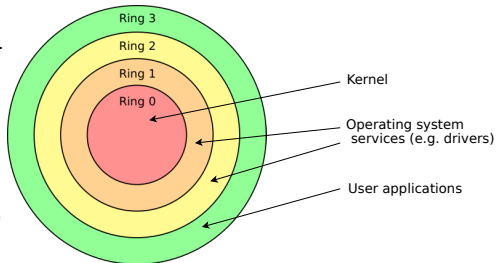
# Operating system services

# Privilege levels

Access to devices, particular data, or some CPU instructions may be *protected* by hardware – only sufficiently privileged code (e.g. kernel code) may access them.

(Why? Suppose all user applications could directly access the disk hardware at any time. The filesystem would be in danger of becoming corrupted. The OS manages orderly access to the hardware.)

For instance, Intel's processors provide 4 privilege levels, conceptualized as rings, where inner rings are the most “trusted”, and outer rings the least.



# Privilege levels

A user application is normally executed at a low level of privilege, and is prohibited from accessing or modifying the memory of other programs, or resources belonging to inner rings; attempting to do so triggers a particular type of *fault* (which can be thought of as a sort of “exception”), e.g. a **general protection fault**.



# System calls

**System calls** constitute the “API” of an operating system kernel – they are the programmatic way to request a service from the kernel.

They allow code running in one of the outer levels (user programs) to obtain a service from one of the inner levels.

An example system call:

the **open** system call on Unix-like systems opens a file for reading or writing.

```
int open(const char *pathname, int flags, mode_t mode);
```

# System calls

From a programmer's point of view, system calls “look” like functions; however, rather than having a normal function body, they typically are implemented as assembly code routines, which do the following:

- ▶ store all the information the kernel needs to provide the requested service in a fixed location
- ▶ execute a “software interrupt”, which causes the kernel to jump to an “interrupt handler”, which examines the information provided
- ▶ the kernel executes some fragment of kernel code that provides the requested service
- ▶ control is then returned to the program that requested the service.

(For more details, refer to e.g. <https://www.cs.montana.edu/courses/spring2005/518/Hypertextbook/jim/index.html> or any operating systems textbook.)

# Dangerous C functions

If you invoke the command `man gets`, you will see the following

## NAME

`gets` — get a string from standard input (DEPRECATED)

## SYNOPSIS

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

```
char *gets(char *s);
```

## DESCRIPTION

Never use this function.

# Dangerous C functions

## BUGS

Never use `gets()`. Because it is impossible to tell without knowing the data in advance how many characters `gets()` will read, and because `gets()` will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use `fgets()` instead.

For more information, see CWE—242 (aka "Use of Inherently Dangerous Function") at <http://cwe.mitre.org/data/definitions/242.html>

And if you try to compile code containing `gets`, `gcc` will tell you  
warning: the 'gets' function is dangerous and should not be used.

# Why is `gets` still around?

The C standard tries to be conservative and backwards compatible.  
Rather than *removing* `gets`, it just says you shouldn't use it.

# Usability of `gets`

[Rusty Russell](#) (an Australian Linux kernel contributor) proposed a [rating scheme](#) for APIs ranging from +10 (“It’s impossible to get wrong”) to -10 (“It’s impossible to get right.”).

- 10. It’s impossible to get wrong.
- 9. The compiler/linker won’t let you get it wrong.
- 8. The compiler will warn if you get it wrong.
- 7. The obvious use is (probably) the correct one.

...

- 7. The obvious use is wrong.
- 8. The compiler will warn if you get it right.
- 9. The compiler/linker won’t let you get it right.
- 10. It’s impossible to get right.

The `gets` function falls firmly into the “-10” level.

So what’s the issue?

# gets

The signature for `gets` is:

```
char *gets(char *s);
```

It reads a line of input from the standard input stream. The idea is that you pass it the address of a **buffer** (array) into which it should copy the line it read.

Here's an example of use:

```
#define BUFSIZE 512
// ...
char buf[BUFSIZE];
printf("Please enter your name and press <Enter>\n");
gets(buf);
```

# gets

```
#define BUFSIZE 512
// ...
char buf[BUFSIZE];
printf("Please enter your name and press <Enter>\n");
gets(buf);
```

The problem is that there is *no* way of telling `gets` how big the buffer `buf` is. If there are more than 512 characters on the line being read, `gets` doesn't stop – it just keeps copying characters into memory, past the end of `buf`.

As we saw when we discussed pointers, this is *undefined behaviour* – at this point, there are no guarantees about what the program will do.



# Morris worm

The flawed behaviour of `gets` was famously used in 1988 by Robert Tappan Morris, a graduate student at Cornell, who created a “worm” program intended to slowly traverse the whole Internet and measure its size.

Due to coding errors on Morris’s part, the worm created new copies as fast as it could, and infected machines became overloaded; Morris’s “worm” brought down most of the Internet.

One of the ways the worm propagated was by exploiting a vulnerability in server programs that used the `gets` function.

# buffer overflows

```
#define BUFSIZE 512
// ...
char buf[BUFSIZE];
printf("Please enter your name and press <Enter>\n");
gets(buf);
```

So what will be sitting in memory after `buf`?

`buf` here is a local variable, sitting in the current stack frame. After it come other local variables, so those will get overwritten; and then the *return address*, the location in memory to go to once the current function has finished; and then the parameters passed to the current function.

# buffer overflows

```
#define BUFSIZE 512
// ...
char buf[BUFSIZE];
printf("Please enter your name and press <Enter>\n");
gets(buf);
```

If you're sending a message to some program that uses `gets`, and you know the structure of its stack frame, you can deliberately overwrite the return address so that execution jumps to code of your choosing (known as “smashing the stack”).

In fact, the data you send could include instructions for executing some arbitrary program (e.g. the shell), and you could force the program to jump to the instructions you just wrote.

# buffer overflows

At least, that's how the stack could be exploited at the time the Morris worm was written.

On modern machines, there are several protections in place against this sort of attack:

- ▶ stack canaries
- ▶ address-space layout randomisation (ASLR)
- ▶ write XOR execute permissions
- ▶ source fortification

More on these in future lectures!

# References

- ▶ Karger, P. A., and R. R. Schell. “Thirty Years Later: Lessons from the Multics Security Evaluation.” 18th Annual Computer Security Applications Conference, 2002. Proceedings., IEEE Comput. Soc, 2002, pp. 119–26,  
<https://doi.org/10.1109/CSAC.2002.1176285>.