Introduction to C

Systems Programming

History of the C language (1/3)

- Developed at Bell Laboratories in **1972** by Dennis Ritchie
  - Many of its principles and ideas were taken from the earlier language **B** and B's earlier ancestors **BCPL** and **CPL**.

- **CPL** (Combined Programming Language) supported both high level, machine independent programming and control over the behavior of individual bits of information – but was too large for use in many applications.

- In 1967, **BCPL** (Basic CPL) was created as a scaled down version of CPL

- In 1970, Ken Thompson, while working at Bell Labs, developed the **B language**, a scaled down version of BCPL written specifically for use in systems programming.

- Finally in 1972, a co-worker of Ken Thompson, Dennis Ritchie, returned some of the generality found in BCPL to the B language in the process of developing the language we now know as **C**.

History of the C language (3/3)

- In 1983, the American National Standards Institute (ANSI) formed a committee, X3J11, to establish a standard specification of C.
  - In 1989, the standard was ratified as ANSI X3.159-1989 "Programming Language C." This version of the language is often referred to as **ANSI C**, **Standard C**, or sometimes **C89**.
- In 1990, the ANSI C standard (with a few minor modifications) was adopted by the International Organization for Standardization (**ISO**) as ISO/IEC 9899:1990.
  - This version is sometimes called **C90**. Therefore, the terms "C89" and "C90" refer to essentially the same language.
- The “latest” standard to address the C language was ISO 9899:2011, issued in 2011.
  - This standard is commonly referred to as "**C11**" It was adopted as an ANSI standard in December 2011.
- C17 is only an error correction of C11 – no relevant changes.
  We will use C17 in this course.

C compared to Assembly

- One level of abstraction “above” assembly
- Language constructs that immensely facilitate **structured programming**
  - E.g. loops
- Automatic **stack** management
- Integrated **heap** management
  - Think of this as a sophisticated version of the memory allocator in the last assembly example
  - But **memory management** is still the responsibility of the **programmer**!
- Strongly **typed**
  - But with **unchecked** type **conversions**!
- **Source-level portability** between processor architectures and operating systems
- The **standard C Library**
  - And many other standards-compliant libraries (e.g. POSIX)

C compared to Java

- Both **high-level** and **low-level language**
- Better control of low-level mechanisms
- **Performance** better than Java
  - But this is not a universal truth any more!
- **Java hides many details** needed for writing OS code

But:

- Not object oriented
- **Memory management** responsibility
- Explicit **initialization** and **error detection**
- More room for **mistakes**
- C exposes many details only needed for writing OS code
- Runs on almost all and even very small/slow CPUs

Simple example

hello.c

```c
#include <stdio.h>

int main(void) {
    /* print out a message */
    printf("Hello World. \n \t Not again! \n");
    return 0;
}
```

Compile (and link):

```
$ gcc hello.c -o hello
```

Run:

```
$ ./hello
Hello World.
        Not again!
```

Summarizing the example

- `#include <stdio.h>`
  - Include header file “stdio.h”
  - No semicolon at end
  - Small letters only – C is case-sensitive

- `int main(void) { . . . }`
  - Program entry point
  - Is the first (user) code executed

- `printf("/* message you want printed */");`
  - Prints a message
  - `\n` = newline
  - `\t` = tab
  - `\` in front of other special characters within printf
    - `printf("Have you heard of \"The Rock\" ? \n");`

The C Compiler

- A compiler translates source code directly into assembly language or machine instructions
  - The eventual end product is a file or files containing **machine code**

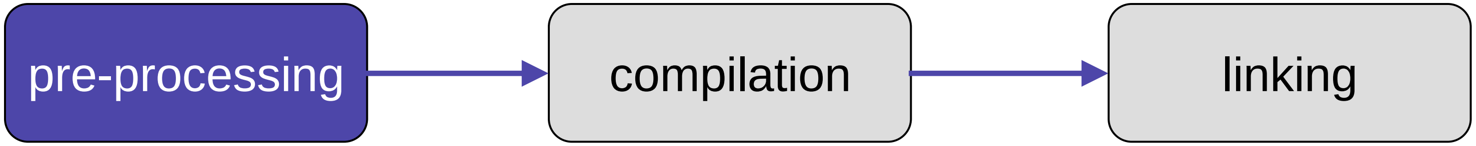
- Some languages (such as C and C++) are designed to allow pieces of a program to be compiled independently
  - These pieces are eventually combined into a final executable program by a tool called the **linker**
  - This process is called **separate compilation**
    - Note: This is different from creating/using libraries!
  - Why is this a useful feature?

- Modern compilers can insert information about the source code into the executable program. (gcc option **-g**)
  - This information is called **debug information** and is used by **source-level debuggers**

The compilation process (1/3)

- In C, compilation starts by running a **preprocessor** on the source code
  - » The preprocessor is a simple program that replaces patterns in the source code with other patterns the programmer has defined (using **preprocessor directives**)
  - » These directives are used, among other things, to save typing, to increase code readability, to control inclusion of header files, etc
  - » The pre-processed code is often written to an intermediate file
  - » Example (compare to assembly directives): **constant declaration**

```
#define BUFFER_SIZE 256
```

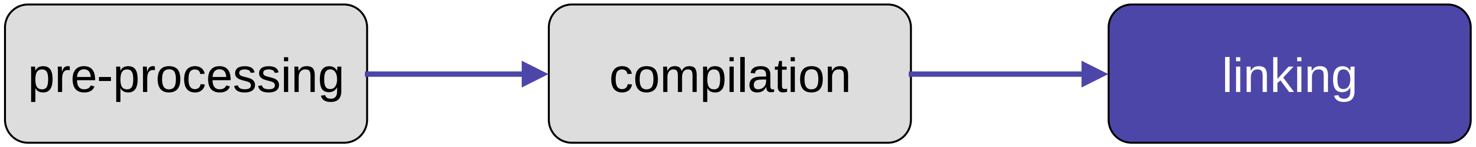


The compilation process (2/3)

- Compilers usually do their work in two passes
  - » The first pass **parses** the pre-processed code into a "parse" tree
  - » In the second pass, the **code generator** walks through the parse tree and generates either assembly code or machine code for the nodes of the tree.
  - » If the code generator creates assembly code, the **assembler** must then be run.
  - » The end result in both cases is an **object module** (a file that typically has an extension of **`.o`** or **`.obj`**).

The compilation process (3/3)

- The linker combines a list of object modules into an **executable program** that can be loaded and run by the OS
  - When a function in one object module makes a reference to a function or variable in another object module, the linker **resolves these references**
  - The linker also makes sure that all the external functions and data you claimed existed during compilation **do exist** exactly **once**
  - Linker also adds a special object module to perform start-up activities
- The linker can search through special files called **libraries** in order to resolve all its references
  - A library contains **a collection of object modules** in a single file



Outline of further topics

- Built-in data types and literals
- Control flow
- Operators and precedence
- Type conversion
- Arrays
- Strings
- Composite data types

Hello World revisited

hello.c

```
#include <stdio.h>

int main(void) {
    /* print out a message */
    printf("Hello World. \n \t Not again! \n");
    return 0;
}
```

Which data types can you find in this example?

» Number: `0`
  - Integer
  - Literal

» String: `"Hello World. \n \t Not again! \n"`
  - Sequence of characters (like in assembly)
  - Literal

» (Type of the) Return value: `int`

C data types (1/3)

- Five important data types in C90
  - **void** associated with no data type
  - **char** character
  - **int** integer
  - **float** floating-point number
  - **double** double precision floating-point number

- Added in C99
  - **long long** (min. 64 Bits)
  - **uint8_t/int32_t** and similar types with defined sizes
  - **bool** (boolean type)
    - Also adds **true**, **false** as boolean literals
    - requires **#include <stdbool.h>**

- Added in C11
  - **char16_t**, **char32_t** (Unicode UTF-16/-32 text)

C data types (2/3)

- **Type modifiers**
  - Several of the basic types can be modified using **signed**, **unsigned**, **short**, and **long**
  - When one of these type modifiers is used **by itself**, a data type of **int** is assumed.
  - A complete list of **possible data types** follows:

```
char             unsigned char          signed char

int              unsigned int           signed int
  short int      unsigned short int     signed short int
  long int       unsigned long int      signed long int

float

double
  long double
```

C data types (3/3)

Note: Values are for 64 Bit Linux!

Type	Bytes	Range
char, signed char	1	-128 .. 127
unsigned char	1	0 .. 255
short, short int, signed short, signed short int	2 (>=2)	-32768 .. 32767
unsigned short, unsigned short int	2 (>=2)	0 .. 65535
int, signed, signed int	4 (>=2)	-2147483648 .. 2147483647
unsigned, unsigned int	4 (>=2)	0 .. 4294967295
long, long int, signed long, signed long int	8 (>=4)	-9223372036854775808 .. 9223372036854775807
unsigned long, unsigned long int	8 (>=4)	0 .. 18446744073709551615
long long, long long int, signed long long, signed long long int	8 (>=8)	-9223372036854775808 .. 9223372036854775807
unsigned long long, unsigned long long int	8 (>=8)	0 .. 18446744073709551615
float	4	$1x10^{-37}$ .. $1x10^{37}$
double	8	$1x10^{-308}$ .. $1x10^{308}$
long double	16	$1x10^{-4932}$ .. $1x10^{4932}$
void	1	-
void*	8	0x00..00 – 0xff..ff

Additional data type facts

- Integer **lengths** (in bytes) in C follow these rules:
  - **`char ≤ short ≤ int ≤ long`**
- The sizes given in the preceding table are **indicative only –** the actual values are **platform- and compiler-specific**
  - But there are definitions in the libraries which **are** precise (e.g., **`int64_t`** designates an exact-width 64-bit integer; see C99)
- To determine the size of any variable type in bytes you can use the **`sizeof`** operator:

```
printf("Size of int is %d\n", sizeof(int));
```

- There was **no boolean primitive until C99**
  - 0 = false
  - Everything else = true
- There is **no byte primitive** in C, so you have to use **`char`**, or more usually **`unsigned char`** instead
- Divide by zero run-time errors and illegal numbers may be returned as: +/- **INF** (infinity) or **IND** (indetermined) or **NaN** (not a number) depending upon the compiler

Literals

- **Characters**
  - Enclosed in **single** quotes (e.g., **'**A**'**)
    - Not zero-terminated, i.e. the example requires one byte for storage
- **Strings**
  - Enclosed in **double** quotes (e.g., **"**str**"**)
    - Zero-terminated, i.e. the example requires **four** bytes for storage
- **Integers**
  - Decimal notation: **160**
  - Hexadecimal notation: **0x**100 (starts with “0x”)
  - Octal notation: **0**100 (starts with “0”)
  - Modifiers for **long** and **unsigned long**: 160**L** and 160**UL**
- **Real numbers**
  - Like integers, followed by the decimal part: 160**.1**
  - If no decimal part is present, the dot must still be added: 160**.**

Before starting

- The thousand and one faces of the main function
  - `[void] main([void])`
    - An explicit void return value is deprecated by most compilers
  - `int main([void])`
  - `[void] main(int argc, char** argv)`
    - As above for explicit void
  - `int main(int argc, char** argv)`

  - "`void`" main functions still return an (OS) exit code – but you can't set it!
  - In the slides for brevity we mostly use "`main(void)`" – the actual source files use "`int main(void)`" instead
  - Using `-Wall` when compiling also reveals potential problems with the signature of `main`.

- All examples are available as separate programs you should compile and test by yourselves!

Typical includes

- The following files are often/typically included
  - stdio.h: Standard input/output, e.g. FILE, stdin, printf
  - stdlib.h: Standard library, e.g. malloc
  - unistd.h: POSIX standard library, e.g. system call wrappers
    - read, write, chdir, fork
  - errno.h: “errno” “variable”, error numbers and names
    - errno is today mostly a macro and no longer a variable
  - string.h: String functions, like strcpy, strcat, memset, atoi
  - sys/types.h: Various data types, e.g. size_t, time_t
  - stdint.h: Data types of fixed/guaranteed length, e.g. uint8_t

Preprocessor

preprocessor.c

```
#include <stdio.h>

#define DANGERLEVEL 5
    /* C Preprocessor - substitution on appearance.
       Somewhat like Java 'final' */

int main(void) {

    float level = 1;
    /* if-then-else as in Java */
    if (level <= DANGERLEVEL) { /* replaced by 5 */
        printf("Low on gas!\n");
    } else {
        printf("Good driver!\n");
    }
}
```

What if we wrote `#define DANGERLEVEL 5;` ?

Control flow

- Branches – identical to Java (actually: Java copied from C!)
  - **`if`** clause (including statements blocks, **`else`**, and nesting)
  - **`switch`** clause (including **`case`**, **`break`**, and **`default`**)
  - Ternary operator (e.g., **`(num > 5) ? [true branch] : [false branch];`**)
- Loops – again, identical to Java
  - **`for`** loop
  - **`while`** loop
  - **`do`** loop
- **`continue`** and **`break`** – **different** from Java
  - Same syntactic rules as in Java
  - But **no labels**!
- **`goto`**
  - Mostly evil –but some people use it for some specific things

Operators and precedence

<table>
<tr><th>Operator type</th><th>Operator</th><th>Precedence</th></tr>
<tr><td>Primary Expression Operators</td><td>() [] . -> expr++ expr--</td><td>Left to right</td></tr>
<tr><td>Unary operators</td><td>* & + - ! ~ ++expr --expr (typecast) sizeof()</td><td>Right to left</td></tr>
<tr><td>Binary operators</td><td>* / %
+ -
<< >>
< > <= >=
== !=
&
^
|
&&
||</td><td>Left to right</td></tr>
<tr><td>Ternary operator</td><td>?:</td><td>Right to left</td></tr>
<tr><td>Assignment operators</td><td>= += -= *= /= %= >>= <<= &= ^= |=</td><td>Right to left</td></tr>
<tr><td>Comma</td><td>,</td><td>Left to right</td></tr>
</table>

Operators and precedence

- **Precedence/Associativity**
  - If two operators from the same line occur together, they are evaluated in this sequence
  - Example a=b=c → Right-to-left → a=(b=c)
  - Note: This is **NOT** “order of evaluation”
    - There are different rules for this!
    - Can also be undefined, like `i = ++i + i++;`

Undetermined!

Undetermined!

- **The logical operators && and || guarantee short-circuit evaluation**
  - Note: The **binary** operators & and | do **NOT**!
  - If the left operand equals 0, the right operand is not evaluated
    - Important for any side-effects, e.g. x**++**

Type conversions (1/2)

- **Implicit (automatic) type conversions**
  - Happens automatically during the course of evaluating an expression
  - Preserve precision (i.e., are always “widening” conversions)
  - If the result value does not “fit” into the result type, it is silently truncated (i.e., no error message!) → inputs are adjusted, not the output!

  - Implicit conversions always follow the arrows
    `bool` → **`char`** → **`int`** → **`float`** → **`double`**

  - Tiebreaker for types that are otherwise the same width
    `signed` → **`unsigned`**

Type conversions (2/2)

- **Assignment conversions**
  - Happen as part of an assignment
  - They **do not** necessarily preserve precision and **no error** is signaled when truncation occurs
  - Example:

```
int num = 312;
char ch = num; /* what is the value of ch? */
```

- **Explicit type conversions**
  - Like in Java

```
    (type) expression
```

  - You can typecast **from any type to any type**

```
  char c = (char) some_int;
```

  - So be careful!

One-dimensional arrays

- 1darrays.c:

```
#include <stdio.h>

main(void) {

        int number[12]; /* 12 cells, one cell per student */
        int index, sum = 0;

        /* Always initialize array before use */
        for (index = 0; index < 12; index++) {
                number[index] = index;
        }
        /* now, number[index]=index; will cause error: why ?*/

        for (index = 0; index < 12; index = index + 1) {
                sum += number[index]; /* sum array elements */
        }

        printf("sum: %d\n", sum);
}
```

Arrays

- **Array**
  - » Sequence of elements of same type, any type

    `int, float, double, char`…
  - » Fixed, constant length
  - » 0-based access via integer index

```
array[0]
array[intVar]
```

  - » **No length information** → have to remember it
  - » **No range checking** → silent over-/underwriting or -reading possible

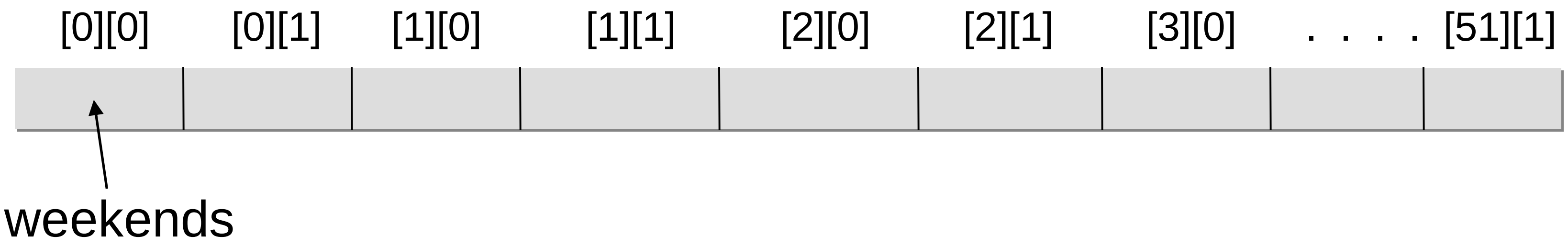
```
int number[12];
printf("%d", number[20]);
```

    Produces undefined output, may terminate, may not even be detected
  - » **Always initialize** before use
- » C99: In functions (and **only** there; not possible for global variables: located in compile-time-sized “data” section !) the length can be set at initialization
  - » `int vla[strlen(in)];`

Arrays with higher dimensions

- **2-dimensional arrays**

```
int weekends[52][2];
```



- **Array access**

```
int points[3][4];
points[2][3] = 12;  /* NOT points[3,4] */
printf("%d", points[2][3]);
```

Strings

- **String**
  - Sequence of characters = character array
  - Terminated by the NUL **character** `'\0'`

```
char name[6];
name = {'C', 'S', '1 ', '0 ', '1 ', '\0'};
       /* '\0'= end of string */

printf("%s", name); /* print until '\0' */
```

- Functions to operate on strings
  - `strcpy, strncpy, strcmp, strncmp, strcat, strncat, strstr, strchr`
  - `#include <strings.h>` at program’s beginning

Composite type creation – we can compose types

- Combining variables with **struct**
- Saving memory with **union**
- Clarifying programs with **enum**
- Aliasing names with **typedef**

Basic structures (1/8)

- A C **structure** is a compound data object
  - Consists of a collection of data objects of (possibly) different types
  - Can be thought of as a private or user defined data type

- In C we can declare such objects by:
  - Defining their internal structure via a **”template”** and
  - Declaring a **tag** to be associated with them

- Given both the declaration and the associated tag, it is only necessary to use the tag when declaring actual instances of structures

- The C keyword **`struct`** is used to indicate that structures are being defined and declared

Basic structures (2/8)

```
struct date /* the tag */ {

    /* start of template */

    int day;    /* a member */
    int month; /* a member */
    int year;  /* a member */
    char dow;  /* a member */

} dates[MAXDAT], today, *next; /* instances */
```

- In the example, **`dates`** is an aggregate of instances of `struct date`, **`today`** is a simple instance of a `struct date` and **`next`** is simply a pointer to a `struct date`.
- Further instances of `struct date` can be declared in the following manner.

```
struct date my_birthday;
struct date end_of_term;
```

Basic structures (3/8)

- In the previous simple example the **structure tag** is **`date`**
  - Tag names conform to the same rules as variable names but belong to a **separate namespace**
  - Because of this, a variable and a tag can have the same name:

    ```
    struct date date;
    ```

- The **template** tells the compiler how the structure is laid out in memory and gives details of the member names
  - A (tagged) template does not automatically reserve any instances

- Structure **member** declarations:
  - Same syntax as ordinary variable declarations
  - Member names like variable names, but in a **separate namespace** (one namespace per structure)
  - I.e. the same name could be used for a structure tag, an instance of the structure and a member of the structure (+of course in other structures)

Basic structures (4/8)

- Structures may be **initialised** in the same fashion as aggregates using initialisers:

```
struct date Christmas = {25,12,1988,3};
```

- Individual **members** of a structure may be referred to, as shown in the following examples

```
dates[k].year
today.month
(*next).day
```

- The **. (dot) operator** selects a particular member from a structure. It has the same precedence as **()** and **[]** which is higher than that of any unary or binary operator. Like **()** and **[]** it associates left to right. The basic syntax is

```
<structure name>.<member name>
```

- Another example:

```
#include <stdio.h>

struct birthday {
     int month;
     int day;
     int year;
};

main(void) {
     struct birthday birth; /* - no 'new' needed */
                            /* then, it's just like Java! */
     birth.day = 1;
     birth.month = 1;
     birth.year = 2003;
     printf("I was born on %d/%d/%d",
            birth.day, birth.month, birth.year);
}
```

Member offsets

- Whenever we access structure members, we are actually accessing a typed variable, whose memory location is defined as an *offset*, **relatively to the address of the structure** variable itself
- The offset of structure members can be obtained using the **`offsetof`** macro, from **`<stddef.h>`**, as the following example demonstrates:

```
#include <stddef.h>

struct date {
    int day;
    int month;
    int year;
};

main () {
    printf("offset of date.day: %d\n", offsetof(struct date, day));
    printf("offset of date.month: %d\n", offsetof(struct date, month));
    printf("offset of date.year: %d\n", offsetof(struct date, year));
}
```

```
offset of date.day: 0
offset of date.month: 4
offset of date.year: 8
```

Basic structures (7/8)

- Structures may be assigned, used as formal **function parameters**, and **returned as function return values**
  - » Such operations cause the compiler to generate sequences of load and store instructions that might pose efficiency problems
  - » This can be avoided by using pointers to structures

- There are few actual operations that can be performed on structures as distinct from their members
  - » The only operators that can be validly associated with structures are "=" (simple assignment) and "&" (take the address)
  - » **It is not possible to compare structures for equality using "==", nor is it possible to perform arithmetic on structures**
  - » Such operations need to be explicitly coded in terms of operations on the individual members of the structure

Basic structures (8/8)

```
struct person {
    char name[41];
    int age;
    float height;
    struct {         /* embedded structure (anonymous) */
      int month;
      int day;
      int year;
    } birth;         /* name of anonymous struct instance */
 };

struct person me;

me.birth.year = 1972;
...

struct person class[34];
                /* array of info about everyone in class */

class[0].name = "bar";
class[0].birth.year = 1982;
...
```

Unions (1/3)

- A **`union`** is **syntactically** identical to a **`struct`**
  - Except that the keyword `union` is used instead of `struct`

- The difference between a struct and a union is that **in a union** the **members overlap each other**
  - The name of a structure member represents the offset of that member from the start of the structure
  - In a union **all members start at the same location** in memory

- The members of a union may themselves be structs and the members of a struct may themselves be unions

Unions (2/3)

- A typical **application** is illustrated by the following fragment.
  - If data, in the form of floating point numbers in internal form, is stored in a file, then it is difficult to read the file since all the standard C file handling functions operate character by character
  - The fragment shown below resolves the difficulty by using a union whose two members consist of a character array and a floating point number. It is assumed that a floating point number occupies 8 characters (1 char = 1 byte).

```
union ibf {
    char c[8];
    double d;
} ibf;
...
double values[...];
...
for (i=0; i<8; i++)
    ibf.c[i] = getc(ifp);
values[j] = ibf.d;
```

Unions (3/3)

Example

```
size of union variable: 8
size of double: 8
offset of variable.character: 0
offset of variable.number_int: 0
offset of variable.number_float: 0
offset of variable.number_double: 0
value of variable.double: 30.700000
value of variable.character: 51
```

```c
#include <stddef.h>

union variable {
    char character;
    int number_int;
    float number_float;
    double number_double;
};

int main (void) {
    union variable var;
    printf("size of union variable: %d\n", sizeof(union variable));
    printf("size of double: %d\n", sizeof(double));

    printf("offset of variable.character: %d\n", offsetof(union variable, character));
    printf("offset of variable.number_int: %d\n", offsetof(union variable, number_int));
    printf("offset of variable.number_float: %d\n", offsetof(union variable, number_float));
    printf("offset of variable.number_double: %d\n", offsetof(union variable, number_double));

    var.number_double = 30.7;
    printf("value of variable.double: %f\n", var.number_double);
    printf("value of variable.character: %d\n", var.character);

    return 0;
}
```

Enums (1/4)

- **`enum`** data types are data items whose values may be any member of a symbolically declared **set of values**
- A typical declaration would be:

```
enum days {Mon, Tues, Weds, Thurs, Fri, Sat, Sun};
```

  - » This declaration means that the values `Mon...Sun` may be assigned to a variable of type **`enum days`**
  - » The actual values are `0...6` in this example and it is these values that must be associated with any input or output operations
    - » You can compare to “Mon” or assig “Tue” to something, but printing such a variable will only produce a number, never a string!
      - » Names are resolved on compilation and don’t exist anymore in the executable (and therefore also not at runtime)!
        - » Except if compiling with debug support…

Enums (2/4)

- For example the following program:

```
enum days { Mon, Tues, Weds, Thurs, Fri, Sat, Sun };

main() {
    enum days start, end;
    start = Weds;
    end = Sat;
    printf ("start = %d end = %d\n", start, end);
    start = 42;
    printf ("start now equals %d\n", start);
}
```

produces the following output:

```
start = 2 end = 5
start now equals 42
```

Notice that it is possible to assign a normal integer to an enum data type and there is **no check** made that an integer assigned to an enum data type is within “range”/”defined”

Enums (3/4)

- Few programmers use enum data types
  - A similar result can be achieved by use of **`#define`**, although the scoping rules are different

- A **difference** between the two approaches is, that it is possible to associate numbers other than the sequence starting at zero with the names in the enum data type by including a specific initialisation in the name list; this also effects all following names.

```
enum coins { p1=1, p2, p5=5, p10=10, p20=20, …};
```

All the names except `p2` are initialised explicitly. `p2` is initialised to value immediately after that used for `p1`, i.e .2.

Enums (4/4) - Limitations

- Enums are not real datatypes, but only a kind of integer
  - This means they will be silently cast to each other
  - Perfectly valid and working:

```
enum e1  {A, B, C };
enum e2  {D, E, F };
void f1(enum e1 param);
void f2(enum e2 param);
enum e1 one = A;
f2(one);
```

- Enums do not have their own namespace (unlike structs) so…
  - two enums cannot have elements of the same name:
    - ```
      enum e1 {A, B, C };
      enum e2 {C, D, E, F }; /* Impossible: C reused*/
      ```
  - variables/functions "shadow" enum elements
    - ```
      enum e1 A = B;  /* Works: variable A*/
      A=A;            /* Impossible */
      ```

Type definitions

- With **typedef** we can create synonyms for types

```
typedef int Employees;
Employees my_company;  /* same as int my_company; */

typedef struct person Person;
Person me;             /* same as struct person me; */

typedef struct person *Personptr;
Personptr ptrtome;     /* same as struct person *ptrtome;
              (we'll explain the '*' above soon) */
```

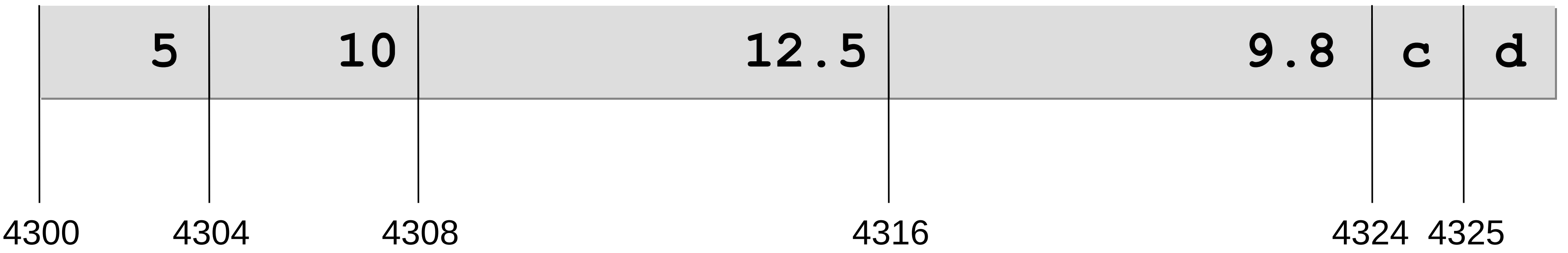
- **Benefits**
  - » Easier to remember
  - » Cleaner code
  - » Very useful for "building up" expressions (coming later)

Outline of further topics

- Variables in memory
- Pointers
- Functions and parameters
- Command line parameters
- Dynamic memory allocation

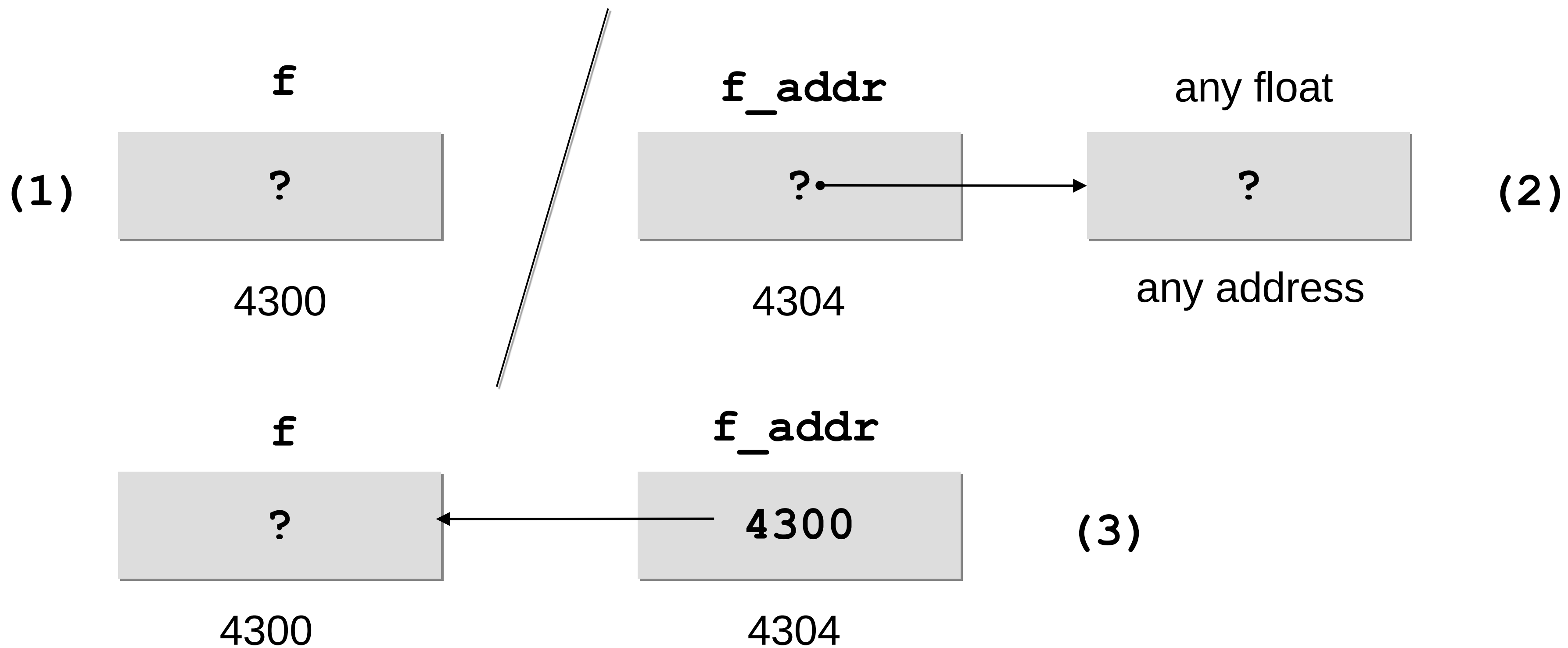
Memory layout and addresses

```
int    x =  5,
       y = 10;

double g = 12.5,
       h =  9.8;

char   c = 'c',
       d = 'd';
```



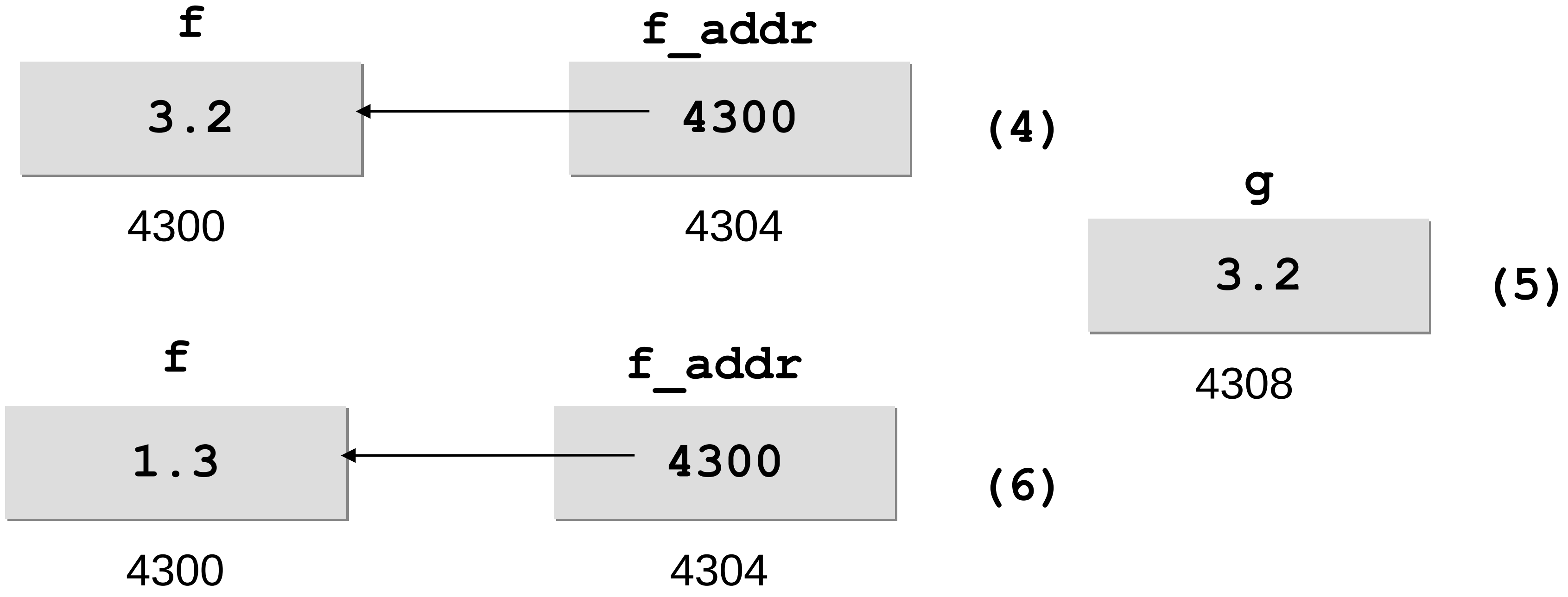
Pointer = variable containing address of another variable

```
(1)      float f;          /* data variable */
(2)      float *f_addr;    /* pointer variable */
(3)      f_addr = &f;      /* & = address-of operator */
```



Pointers made easy – almost! (2/2)

```
(4)    *f_addr = 3.2;       /* * = indirection operator */
(5)    float g = *f_addr; /* g is now 3.2 */
(6)    f = 1.3;             /* g is still 3.2 ! */
```



Why pointer arguments? (1/2)

pointer_args_1.c

```
#include <stdio.h>

void swap(int, int);

main(void) {
        int num1 = 5, num2 = 10;
        swap(num1, num2);
        printf("num1 = %d and num2 = %d\n", num1, num2);
}

void swap(int n1, int n2) { /* passed by value */
        int temp;
        temp = n1;
        n1 = n2;
        n2 = temp;
}
```

Why pointer arguments? (2/2)

pointer_args_2.c

```
#include <stdio.h>

void swap(int *, int *);

main(void) {
    int num1 = 5, num2 = 10;
    swap(&num1, &num2);
    printf("num1 = %d and num2 = %d\n", num1, num2);
}

/* passed and returned by reference */
void swap(int *n1, int *n2) {
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

What' s wrong with this?

pointer_problem.c

```
#include <stdio.h>

void dosomething(int **ptr);

main(void) {
        int *p;
        dosomething(&p);
        printf("%d", *p); /* will this work? */
        printf("%d", *p); /* and now? */
}

void dosomething(int **ptr) { /* passed and returned by reference */
        int temp = 32 + 12;
        *ptr = &temp;
}
```

```
$ gcc pointer_problem.c -o wrong

$ ./wrong
44
6733812
```

Compiles correctly, but crashes or results in … unexpected output.

Passing and returning arrays

pass_and_return_arrays.c

```
#include <stdio.h>

void init_array(int array[], int size);

main(void) {
        int j, list[5];
        init_array(list, 5);

        for (j = 0; j < 5; j++)
            printf("next:%d\n", list[j]);
}

void init_array(int array[], int size) { /* why size ? */
        /* arrays are ALWAYS passed by reference */
        int i;
        for (i = 0; i < size; i++)
                array[i] = 0;
}
```

The `argc` and `argv` parameters

cmd_line_args.c

```
#include <stdio.h>

/* program called with command-line parameters */
main(int argc, char *argv[]) {
        int ctr;

        for (ctr = 0; ctr < argc; ctr = ctr + 1) {
                printf("Argument #%d is -> |%s|\n",
                        ctr, argv[ctr]);
                /* ex., argv[0] == the name of the program */
        }
}
```

Passing / returning a structure

```
/* pass struct by value */
void display_year_1(struct birthday mybday) {
       printf("I was born in %d\n", mybday.year);
}                                 /* - inefficient: why ? */

. . .
/* pass struct by reference */
void display_year_2(struct birthday *pmybday) {
   printf("I was born in %d\n", (*pmybday).year);
                                 /* warning! not just ‘.’,
                                    after a struct pointer */
}

. . .
/* return struct by value */
struct birthday get_bday(void) {
  struct birthday newbday;
  newbday.year = 1971;           /* ‘.’ after a struct */
  return newbday;
}                                /* - also inefficient: why ? */
```

Finally! The pointer operator

- We said before that this is wrong:

```
*pmybday.year      /* wrong */
```

  - That's because the **.** (dot) operator has higher priority than the ***** (star) operator, so it's equivalent to: `*(pmybday.year) /* same; wrong */`

- The correct way of referring to a member of a structure whose address is given is typically:

```
(*next).day  /* correct */
```

- This is so common that the alternative syntax

```
next->day  /* same; correct */
```

is part of the C Programming language. The **-> operator** has the same precedence as the . (dot) operator.

Dynamic memory allocation

Allocation *and de-allocation* in C is explicit

memory_allocation.c

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

int main(void) {
       int *ptr;

       /* allocate space to hold an int */
       ptr = (int*) malloc(sizeof(int));
       if (ptr==NULL) return -1;

       /* do stuff with the space */
       *ptr = 4;

        /* free up the allocated space */
       free(ptr);
       return 0;
}
```

Dynamic allocation of memory for structures

- The dynamic allocation of memory intended to contain structures is rather simple: one only has to **reserve enough space** to hold all the members of a structure

- This, however, contains the implicit assumption that we know how many bytes are required to store a particular structure
  - » Not trivial, because of **alignment constraints**(=restrictions on which kinds of addresses things can begin, e.g. even, multiple of 8/16…)

- Fortunately, **`sizeof`** can be used to determine that size, taking any alignment constraints also into consideration

- Thus it's possible (& recommended!) to use expressions like:

```
struct date *a = (struct date *) malloc(sizeof(struct date));
```

Outline of further topics

- Declarations and definitions
- Source code organization
- Separate compilation
- Storage qualifiers
- Naming rules
- Reserved words

Declarations vs. definitions

- In C declarations and definitions are **not the same thing**
- **Declaration** introduces a name (= identifier) to the compiler
  - Creates an entry in the compiler's list of "things to assign an address"
- **Definition** allocates storage for the name
  - » This meaning applies for both variables and functions
  - » For a variable → space is reserved in memory to hold the data
  - » For a function → the compiler generates code, which ends up occupying storage in memory
- You can declare a variable or a function in many different places, but there **must be only one definition per item in C** (this is sometimes called the ODR: one-definition rule)
  - » This is checked when the linker is uniting all the object modules
- A **definition can also be a declaration**. If the compiler hasn't seen the name *x* before and you define **`int x`**, the compiler sees it as a declaration and allocates storage for it all at once.

Function declarations

- A **function declaration** in C gives the function name, the argument types passed to the function, and the return value of the function:

```
int func1(int,int);
```

- C declarations attempt to **mimic the form of the item's use**. For example, if *a* is an integer the above function might be used in this way:

```
a = func1(2,3);
```

- **Arguments** in function declarations may have names. The compiler ignores the names but they can be helpful as memonic devices for the user what is expected :

```
int func1(int length, int width);
```

Function definitions

- **Function definitions** look like function declarations except that they **have bodies**
- A body is a collection of statements enclosed in braces:

```
int func1(int length, int width) {
    . . .
}
```

- Notice
  - In the function definition the **braces { and }** replace the semicolon
  - The **arguments** in the function definition **must** have names if you want to **use** the arguments in the function body

(No) Function overloading

- **Function overloading** is defining a function with the same name again, but with different parameter lists
  - C does **NOT** support this!

- So this is not possible (in C):

```
int func(int param);
int func(float param);
```

  - Error: Conflicting types for ‘func’

- Notice:
  - » C++ and many other modern languages **do** support this!

Variable declarations

- Variable declarations are not identical to function declarations. For example **this is not a declaration**:

```
int a; /* Wrong! This is a definition */
```

- In the code above is sufficient information for the compiler to create space for an integer called *a* - and that's what happens
- To resolve this problem, a keyword was necessary for C to say "This is only a declaration; it's defined elsewhere."
- This keyword is **`extern`**
  - Note that **`extern`** can mean that the definition is **external** to the file, or that the definition occurs **later** in the file
  - **`extern`** can also be used for function declaration for consistency
- Example of a **proper variable declaration**:

```
extern int a; /* Correct! This is a declaration */
```

Externs

```
#include <stdio.h>

extern char user2line[20];      /* global variable defined
                                   in another file */
char user1line[30];             /* global for this file */
void dummy(void);

main(void) {
 char user1line[20];            /* different from earlier
                                   user1line[30] */
 . . .                          /* restricted to this function */
}

void dummy(){
 extern char user1line[];       /* the global user1line[30] */
 . . .
}
```

Storage classes

- **`extern`**
  - Just a declaration = reference to variable defined elsewhere
  - Scope: block or file
- **`static`**
  - Lifetime: program's run-time
  - Scope: block or file (limited to this .c file; not visible outside)
  - Initialized to 0
  - Implicit for variables defined outside blocks
- **`auto`**
  - Lifetime: block
  - Implicit in blocks; can be left out
  - Uninitialized
- **`register`**
  - For automatic (i.e., local) variables
  - Hint to compiler: local variable; preferably put it into a register

Including headers (1/2)

- Most libraries contain significant numbers of functions and variables. To save work and ensure consistency when making the external declarations for these items, C uses a device called the **header file**.
  - A header file is a file containing the external declarations for a library, and typically has the extension '.h'
  - Library header files are provided along with the libraries themselves

- To declare the functions and external variables in the library, the user simply **includes** the header file. To include a header file, use the **`#include`** preprocessor directive.
  - This tells the preprocessor to open the named header file and insert its contents where the `#include` statement appears
  - `#include` may name a file in two ways: in **angular brackets** (**`< >`**) or in **double quotes** (`" "`)

Programs with multiple files

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

int main(void){
 myproc();
 printf("%d", data)
}
```

hw.c

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

int data;

void myproc(void)
{
 data = 2;
 . . . /* some code */
}
```

mypgm.c

```
void myproc(void);
extern int data;
```

mypgm.h

- Library headers
  - » Standard: < >
  - » User-defined: " "

Separate compilation example

- Compile individual source code files

```
gcc -c mypgm.c
gcc -c hw.c
```

  - Generates mypgm.o and hw.o

- Link object files to executable

```
gcc -o hw mypgm.o hw.o
```

Separate compilation

- Separate compilation is important in **building large projects**
  - The most fundamental tool for breaking a program up into pieces is the ability to create named subroutines or subprograms
  - In C, a subprogram is called a function, and functions are the "atomic" units of code

- To create a program with multiple files, functions in one file must be able to **access functions and data in other files**
  - When compiling a file, the C compiler must know about the functions and data in the other files, in particular their names and proper usage
  - The compiler ensures that functions and data are used correctly
  - This process of "telling the compiler" the names of external functions and data and what they should look like is the **declaration**
  - Once you declare a function or variable, the compiler knows how to check to make sure it is used properly

Including headers (2/2)

- File names in **angular brackets**, such as:

```
#include <header.h>
```

cause the preprocessor to search for the file in the **"include search path"**. The mechanism for setting the search path varies between machines, operating systems, and C implementations.

  - Typical use: standard/OS libraries location

- File names in **double quotes**, such as:

```
#include "local.h"
```

tell the preprocessor to search for the file in an "implementation-defined way." What this typically means is to search for the file **relative to the current directory**. If the file is not found, then the include directive is reprocessed as if it had angular brackets instead of quotes.

  - Typical use: parts of this project/ program

Variable type qualifiers

- **`const`**
  - » **Write-protect** variable
  - » Enforced by **compiler**
  - » Integer constant: `const int five = 5;`
  - » (Variable) pointer to constant integer: `const int * ptr`
  - » Constant pointer to (variable) integer: `int * const ptr`

Only single memory location (but has one!)
#define → copied to every occurrence

- **`volatile`**
  - » Value (in memory) **may change** even if not written to in program
  - » For example by another process or hardware
  - » Prevents certain optimizations by compiler, where variable is not read again (because it wasn't written to in the meantime)

```
static volatile int flag;
void check (void) {
  flag = 1;
  while (flag) { doSomething(); }
}
```

Naming rules

- **Identifier** = name of
  - Variable
  - Function
  - Parameter
  - Template tag of structures/unions/enums
  - Member of structures/unions
  - Type definition
- Can **consist of**
  - Upper- and lower-case ASCII letters
  - Decimal digits
  - Underscore character
- Has to **start** with letter
- Maximum of **31 characters**
- Must **not** be one of the **reserved keywords**

Reserved keywords – ANSI C (C89) and ISO C (C90)

- auto
- break
- case
- char
- const
- continue
- default
- do
- double
- else
- enum
- extern
- float
- for
- goto
- if
- int
- long
- register
- return
- short
- signed
- sizeof
- static
- struct
- switch
- typedef
- union
- unsigned
- void
- volatile
- while

C99

- _Bool / bool
- _Complex
- _Imaginary
- inline
- restrict

What does this C program do? (1/2)

```
#include <stdio.h>

struct list {
    int data;
    struct list *next;
};

struct list *start, *end;

void add(struct list **head, struct list **tail, int theData)
{
    if (*tail==NULL) {
        *head = *tail = (struct list *)malloc(sizeof(struct list));
        (*head)->data = theData;
        (*head)->next = NULL;
    } else {
        (*tail)->next = (struct list *)malloc(sizeof(struct list));
        *tail = (*tail)->next;
        (*tail)->data = theData;
        (*tail)->next = NULL;
    }
}
```

What does this C program do? (2/2)

```
void delete (struct list **head, struct list **tail) {
    struct list *temp;
    temp = *head;
    if (*head==*tail) {
        free(*head);
        *head = *tail = NULL;
    } else {
        temp = (*head)->next;
        free(*head);
        (*head) = temp;
    }
}

int main() {
   start = end = NULL;
   printf("Adding '2'\n");
   add(&start, &end, 2);
   printf("Adding '3'\n");
   add(&start, &end, 3);
   printf("First element: %d\n", start->data);
   printf("Deleting one\n");
   delete(&start, &end);
   printf("New first element: %d\n", start->data);
   /* Memory leak - two malloc but only one free! */
}
```

A few good hints

- Always **initialize** anything before using it
  - » Especially pointers
- Don't use memory pointed to by pointers after **freeing** it
- Don't free pointers **twice**
  - Might now have been reserved by someone else
  - Could now be inside a memory block used by someone
- Don't return a function's **local variables** by reference
- There are no exceptions – so **check for errors** everywhere
- The name of an **array** acts like a pointer, but its value (the address) is **immutable**

- We will look at most of these things in more detail in the coming lectures