NTNU

Norwegian University of
Science and Technology

C in 90 minutes

1. Compilation: What happens behind the scenes
2. Basic blocks and the run-time stack
3. Pointers and dynamic memory management
4. Fundamental and derived types
5. Variable-length argument lists
6. Additional preprocessor control

Why C?

- At their best, programming languages & models provide abstractions which match the problem you are trying to solve
  - OO programming lets you write in terms of structures which map onto "*things*" - colors, invoices, birds and snakes and aeroplanes
  - Functional programming lets you string together functions to be evaluated, in terms of how they combine
  - Relational programming lets you assert a lot of facts, so you can ask the system what else is true as a consequence of these
- *The abstractions of C do not address the problem you are making a solution for – C abstracts the computer which executes that solution*
- That's dandy for us, we are after making our own abstractions, and mapping them onto the computer

Why not C?

- Compiler construction has a plethora of worthwhile abstractions, and would be very well served by a bit of language support
- Our reason for not using any, is that they potentially create magic black boxes for programmers to trust
- Doing this without black boxes
  - Ensures that you get an idea what's inside the box before using it
  - Seriously hampers productivity
  - Restricts us to making a rather tiny and silly compiler
- *In terms of dragon metaphors, we're picking wooden swords to practice pummeling Barney the purple dinosaur*
- Should your line of work ever pit you against any actual dragons which breathe fire, make sure to bring something sharper

Starting somewhere: Hello, world! (in a file “hello.c”, or similar)

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

int
main ( int argc, char **argv )
{
    printf ( “Hello, world!\n” );
    exit ( EXIT_SUCCESS );
}
```

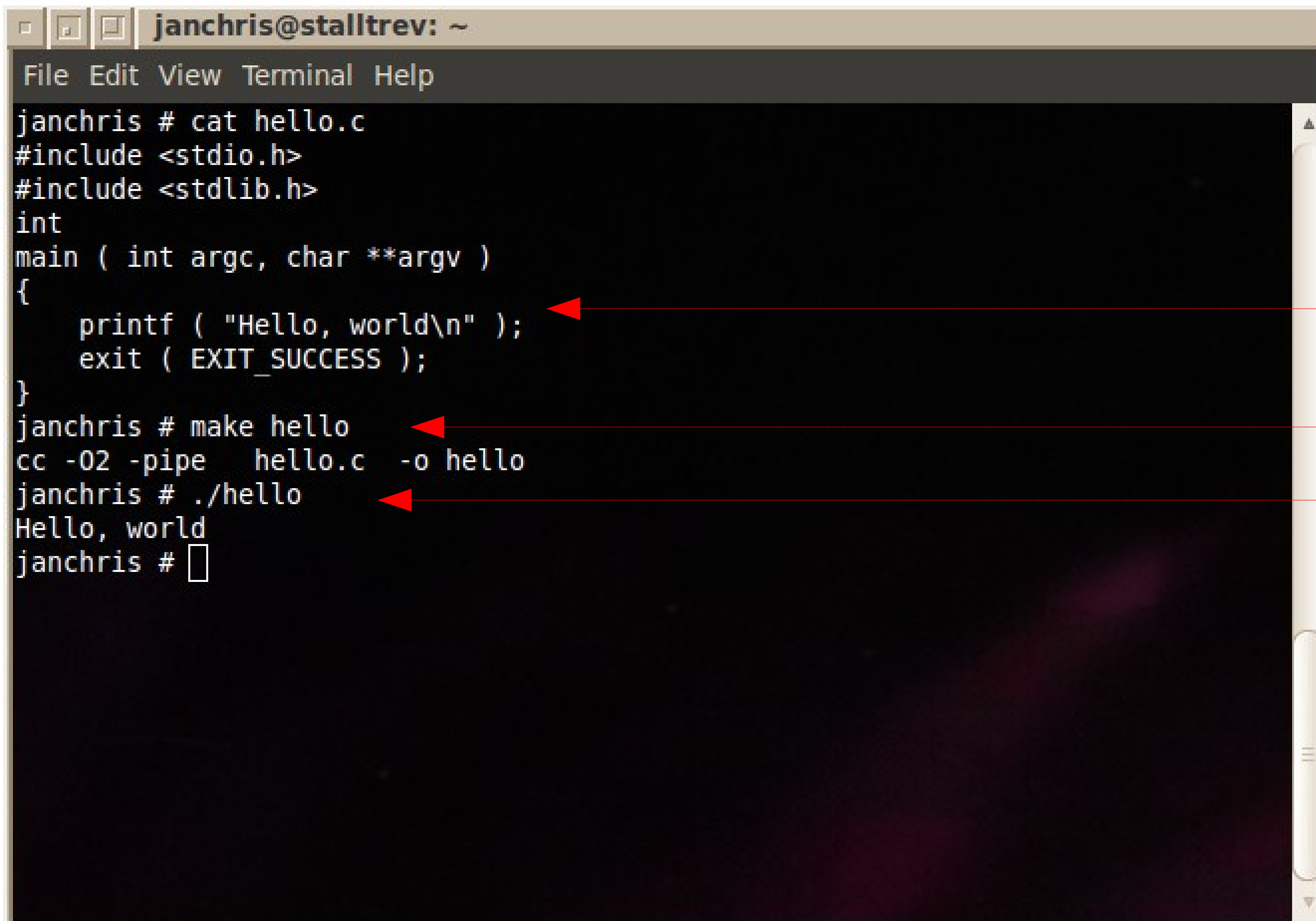
Preprocessor directives (→ #include lines)

Starting point (→ main)

Two function calls (→ printf, exit)

(Wow!)

Making something happen



Program source

Compile & link

Run

The assumptions made

- That you can run a plain text editor
- That you can store its output on a system running some kind of UNIX-flavored operating system
- That you can log in to and interact with a command shell on that machine
- That you are familiar with **make**

- To some this is already 2nd nature, to others it looks like a blast from the 1970s
- To catch everyone, I'll dissect the practical concerns at great speed...

The tool chain

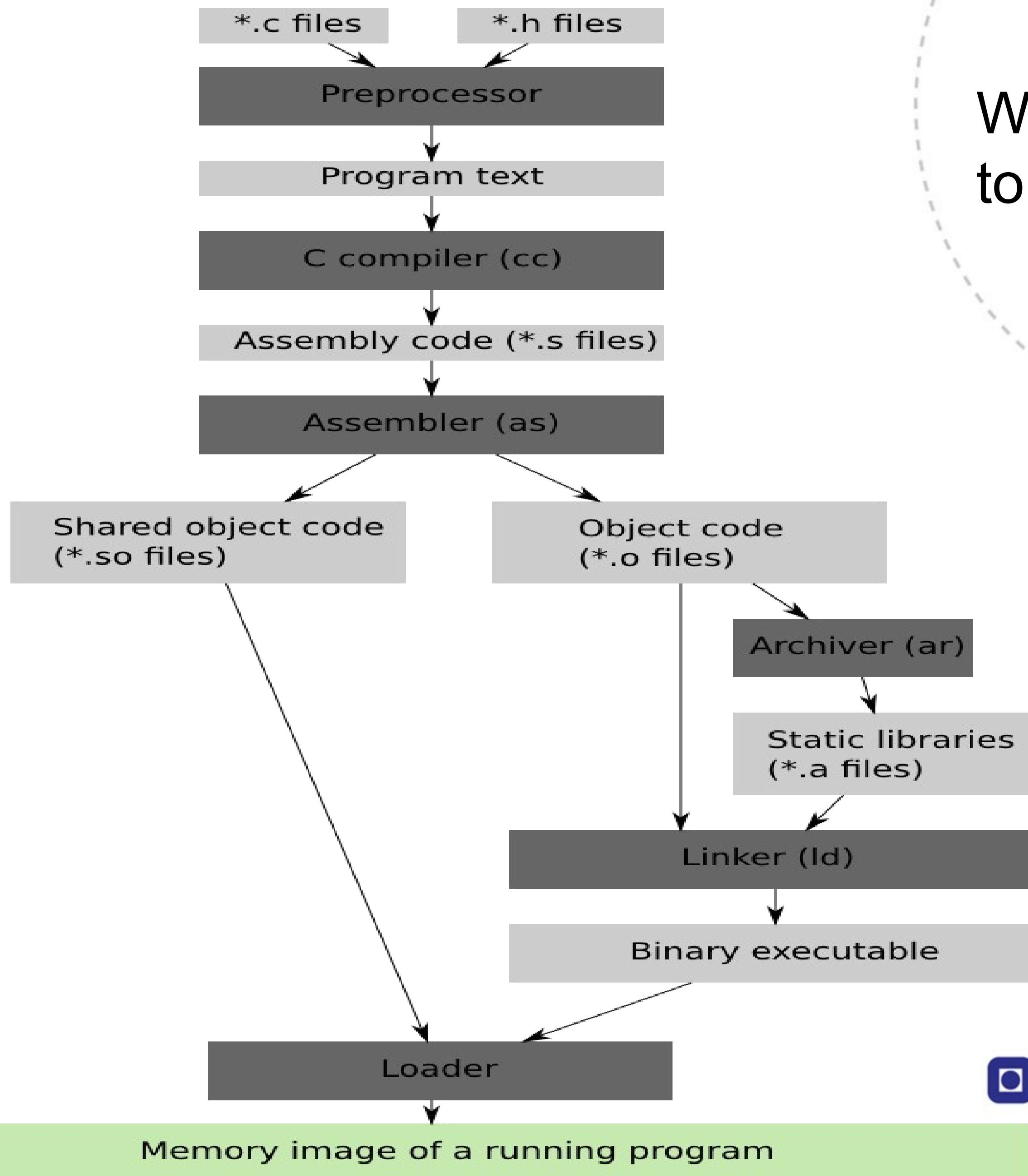
- We work with standard POSIX tools - not because they are perfect, but because
  - they're available on every operating system I've heard of (mostly by default, otherwise by a reasonably small installation)
  - even if you don't want them on your own machine, it's easy to use them remotely on NTNUs systems
- From the bottom up,
  - 'ld' is the linker, which produces binary executables from object code
  - 'as' is the assembler, which produces object code from assembler code
  - 'cc' is the C compiler, which produces assembler code from C code
- These are standard symbolic names, and refer to whatever tools are default on the system you're using
- I use the GNU toolchain (all of the above come from GCC), but we're not doing anything very platform-specific, so things should be pretty portable.

Just to get going

- If you don't have a convenient system handy, cc and friends can be found on **login.stud.ntnu.no**
- You can have SSH shells from windows, tiny program download from http://www.putty.org/
- You can transfer files through SAMBA (“map network drive”), or edit them directly through the shell ('nano' is a pretty humane screen-editor available on login.stud, documentation at http://www.nano-editor.org/)
- None of this is particularly hard, but it isn't perfectly intuitive to everyone the first time.
- *If you can't find your way, ask. Installing a 100 megabytes of colorful buttons will not solve the problem.*

(Corollary: if you *can* find your way, feel free to use whatever IDE you know and love, but don't rely on it being there)

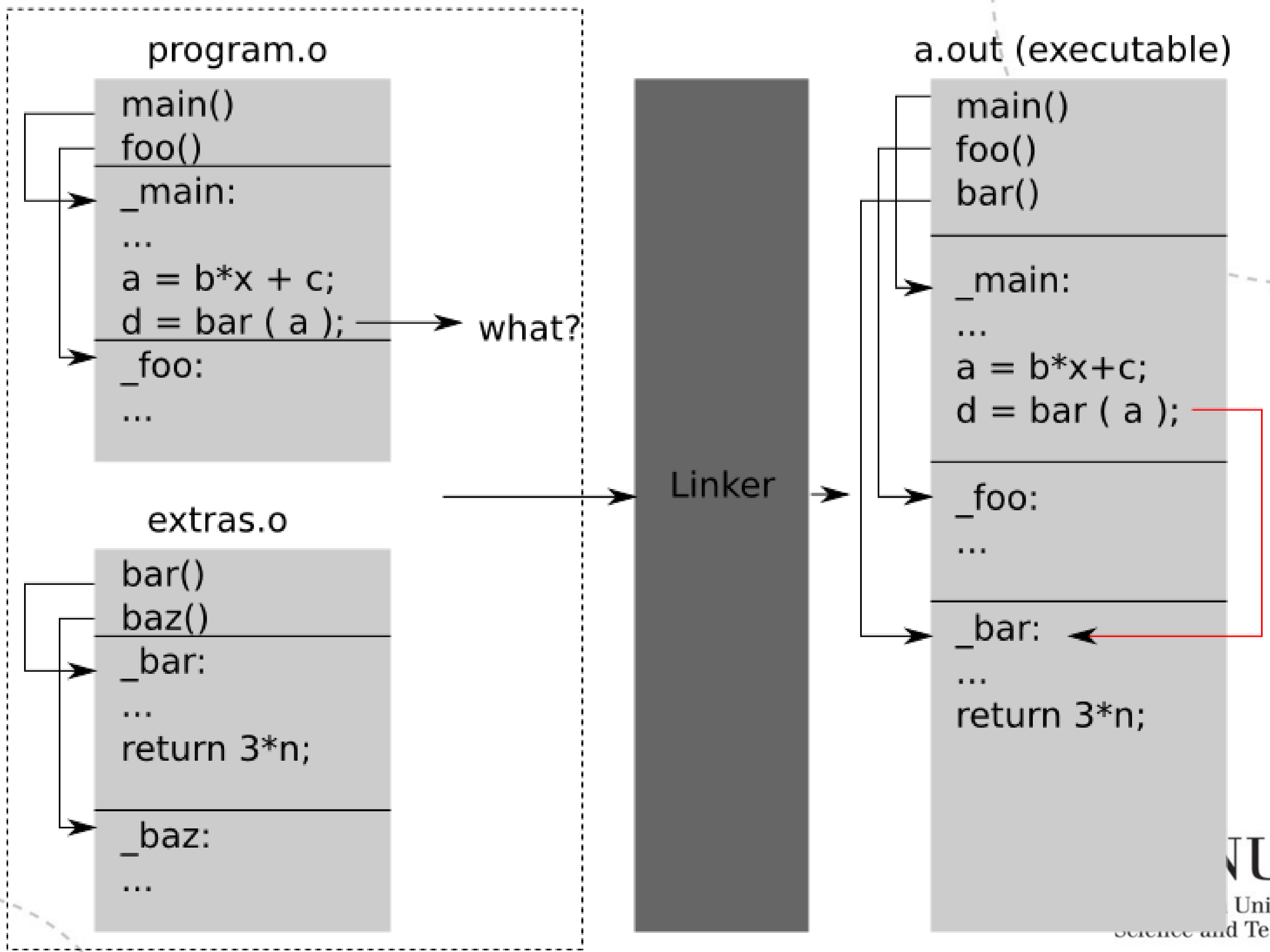
What happens to a program?



NTNU
Norwegian University of Science and Technology

That's a lot of stuff...

- The key takeaway is to look at C source files as recipes for object code we feed to the linker (and the loader, but we won't dabble with that).
- So,
  - what is object code?
  - what does the linker do?
- Object code is a file full of machine instructions, where all the addresses are relative inside the file itself
- The linker takes several of these and glue them together, making references from one point into the others where necessary.


What does the linker do?
('ld program.o extras.o')
program.o
main()
foo()
_main:
...
a = b*x + c;
d = bar (a);
what?
_foo:
...
extras.o
bar()
baz()
_bar:
...
return 3*n;
_baz:
...
Linker
a.out (executable)
main()
foo()
bar()
_main:
...
a = b*x+c;
d = bar (a);
_foo:
...
_bar:
...
return 3*n;

So, to feed the linker...

- ...each translation unit must define
  - Names of functions available to the outside world (function decl.)
  - Names of data available to the outside world (globals, externals)
  - Anything not named in the head of the object code will only be accessible internally in the file
- This is what passes for encapsulation & interfaces in C, and program structure tends to reflect it in the way it's chopped up into independent files
- It's not perfectly 1-1, if you feed multiple source files to the compiler it produces 1 object code from the lot
- Maintaining locality by file keeps dependencies simple
- Whenever 'main' is defined, it's linked from the O/S dependent start-the-program-code.

NTNU
Norwegian University of Science and Technology

Hello, world line by line

- *"#include <stdio.h>"* pastes in the standard I/O functions ("printf" here, which outputs characters)
- *"#include <stdlib.h>"* pastes in the standard library functions ("exit" here – we could have done without it, but I tend to write it for clarity)
- *"int main (..."* defines the address where an executable should begin execution, so the linker can find it
- *"printf (..."* defines a point where the linker has to dig out the *printf* object code of the std. libraries, and glue in a reference
- *"exit (..."* does the same for the exit function

Header files

- #include <stdio.h> does make the preprocessor dig out a text file called 'stdio.h', and put all its contents where the directive was. You could find stdio.h yourself, and do this with copy/paste.
- The '<>' means “look for it in the default path for system things” - writing *#include “myfile.h”* instead would make the preprocessor look around the directory where the rest of *your* code lives.

- stdio.h doesn't actually contain any code for *printf* – it just has a function definition without a body, something like

  ```
  int printf ( const char *fmt, … );
  ```

  which tells the compiler that *yes, there is a function like this*, its first parameter is a constant string, and the linker should be trusted with finding the actual object code.
- Thus, printf can be compiled once and for all, while only its *interface* is run 10.000 times through the compiler (and you don't need the source for printf itself).

NTNU
Norwegian University of Science and Technology

The next level up

- At this point, we have dismantled the genesis of a runnable binary, roughly
- Object code is a system level *lingua franca*, compiled languages are just various notations for specifying it, differing in which conveniences they provide
- If you know what the object code which comes from a given language looks like, you can combine parts from different ones
- We take the C language from here

Statements and declarations

- The basic syntax of C looks a lot like Java (or rather, the other way around)
- Everyone will have written some programs, I'm not spending time on basics of
  - What is a variable
  - What is a constant
  - What is an assignment
  - What is a condition
  - What is a function
  - What is a loop
  - What is recursion
  - What is yadda yadda from programming 101
- Let's call them statements and declarations

Statements combining statements

- When you have some statements in C, a *{ basic block }* combines them into a single statement.
- i.e.

```
a = b + c;
c += 42;
```

  is two statements, equally well written as

```
{
    a = b+c;
    c += 42;
}
```

  which is one basic block, and is itself a statement.
- If, for, while, etc. are followed by a statement, so

  `if ( a!=0 ) a = b*x;` ↔ `if ( a!=0 ) { a = b*x; }`

  are practically equivalent.
- Thus, we can make loops and conditionals which contain more than one statement, the compiler just sees a single statement fitting where it should.

NTNU
Norwegian University of Science and Technology

What's special about that?

- The reason for highlighting the basic block, is that it is the building block of local context.
- Witness:

```
int a = 1, b = 0;
{
    int a = 64;
    b = a – 32;
}
printf ( “ a is %d, b is %d\n”, a, b );
```

- This code will print *“a is 1, b is 32”*; the 'a' declared inside the basic block overrides the exterior a, but is gone when the block ends.

Very Generally^{TM}

- C programs consist of functions and data
- A function is no more than a basic block with a name and a few decorations
  - parameters allow a bit of the local context to be put in
  - return value allows a bit of it to be taken out
- Basic blocks nest inside each other, but only the top-level ones can be given names
- This gives us that execution is just a bunch of basic blocks passing control between each other, and their names define the lookup table at the head of the object code file

NTNU
Norwegian University of Science and Technology

Referring to global data

Declaring x globally makes it visible to all functions

```
int x = 32;
int main ( int argc, char **argv )
{
    print_x_value();
}

void print_x_value(void)
{
    printf ( “x is %d\n”, x );
}
```

- This works because 'x' goes in the object code's name table.

Referring to external data

main_program.c:

```
int x = 32;
int main ( int argc, char **argv )
{
    print_x_value();
}
```

extra_functions.c:

```
extern int x;
void print_x_value(void)
{
    printf ( "x is %d\n", x );
}
```

This tells the linker that there is a global x to look up elsewhere

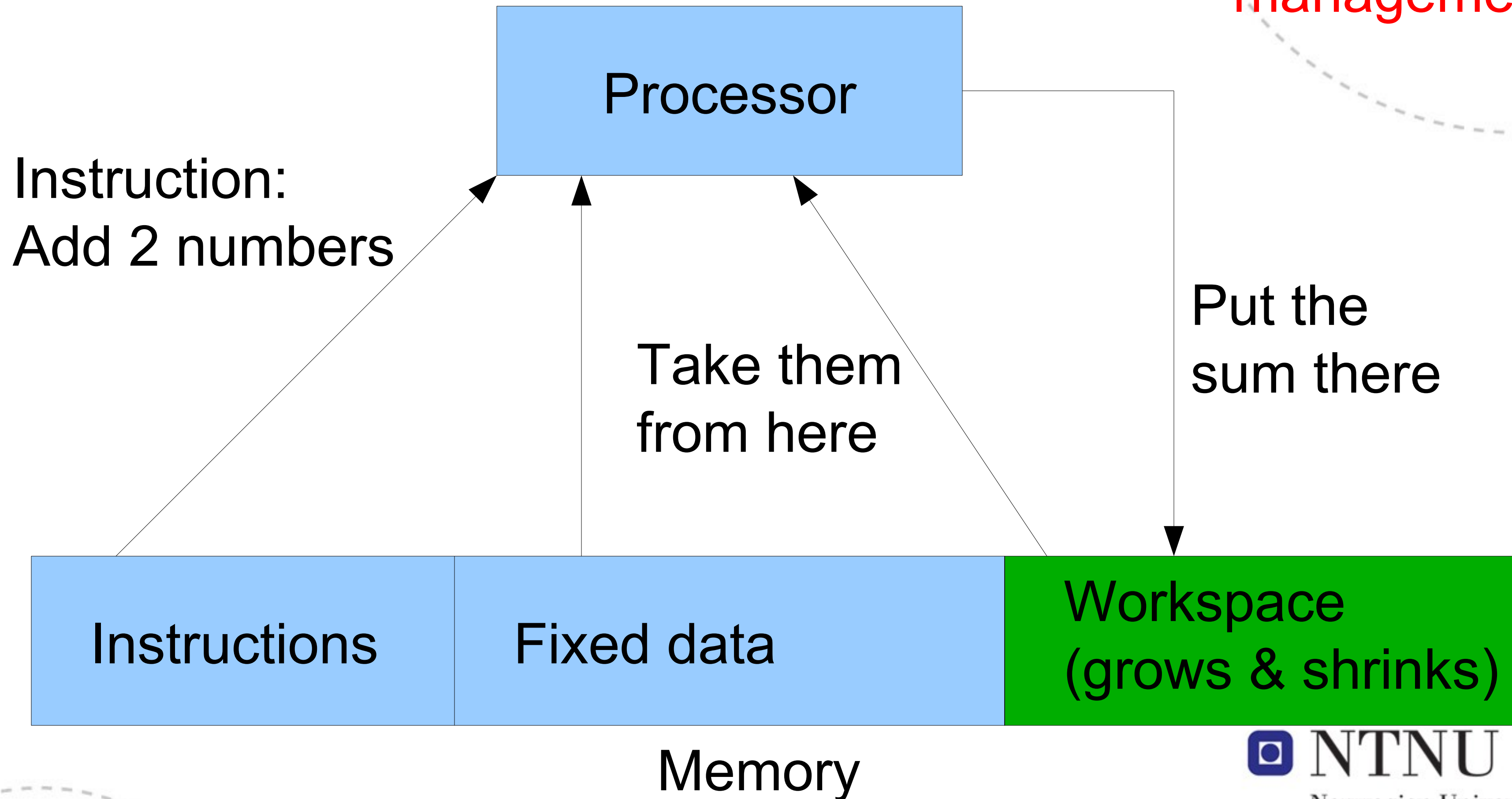
- If *main_program.c* declared 'static int x = 32;', its name would not appear in the object code, and the linker would resolve 'x' to something else (or fail trying).

Managing the global namespace

- The same goes for functions:
  - By default, they find their way into the object code, which makes them callable from other object code
  - If you declare them *static*, their names disappear forever at compile time, as every call inside their file are resolved once and for all.
- This is all that's needed to define programs which already know everything about their memory:
  - Declare a bunch of global variables (and arrays)
  - Declare a family of functions which manipulate them
  - Compile, link and go
- Most *useful* programs don't know how much they will need until they run, though.
- The compiler can't deduce that, so the code itself must be written to handle all things dynamic.

Very Simple Computer
(with dynamic memory)

This part needs management



Stack & heap

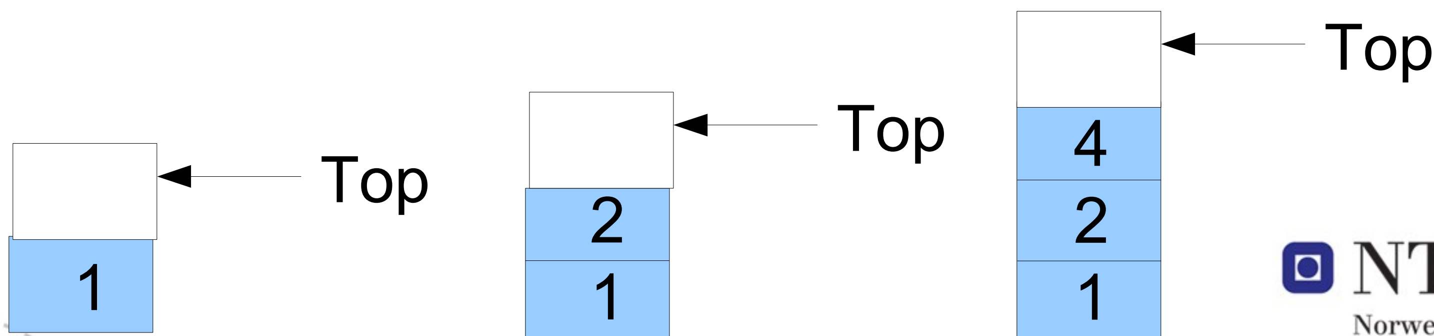
- The variable workspace indicated on the last foil divides into *stack* and *heap* memory.
- It's really just a flat, boring bunch of addresses – we divide it in two because we want to use it for two different things
- Thing #1 is *stack space*, where the compiler can write the memory management code from looking at basic blocks
- Thing #2 is *heap space*, which the compiler knows nothing about, so the programmer gets to take care of the whole enchilada

The run-time stack

- Most CPUs have a sort-of special register which tracks the top of a stack structure in memory
- What makes it special is that it will increment/decrement as a side-effect of *push* and *pop* operations, so a sequence of operations

    push 1
    push 2
    push 4

  will cause something like this in memory:



The run-time stack II

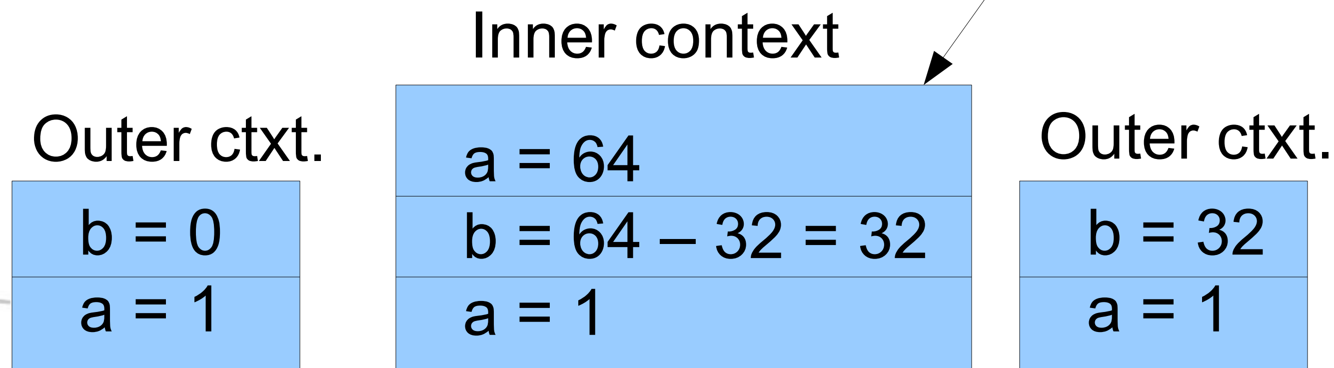
- Somewhat obviously, three *pop* operations will retrieve the numbers 4, 2, 1 in that order, and leave the top-of-stack register where it was.
- When a program starts, the bottom of the stack is placed at the beginning of a “large enough” tract of usable memory, so that an arbitrary amount of pushing and popping can claim and recycle memory as needed, while the program is running.
- This is the mechanism which is used for the local context of a basic block (and thus by extension, also for the local variables in a function).

Basic blocks & run-time stack

Looking at this one again:

```
{
    int a = 1, b = 0;
    {
        int a = 64;
        b = a – 32;
    }
    printf ( “ a is %d, b is %d\n”, a, b );
}
```

What a block adds to the top of the stack is removed when it ends



NTNU
Norwegian University of Science and Technology

Basic blocks & run-time stack

- The names of variables are looked up by the first match from the top down, which is what gives the effect / scope-rule that the inner declaration overrides the outer
- The reason stack space can't be computed to an exact fit at compile time, is that functions are basic blocks.
- Each function call eats a bit of stack space, and you can't determine ahead of time how many function calls there will be.
- Because of its dynamic nature, you can also put variable-length arrays on stack

  (globally declared arrays require constant size)

Pointers, in principle

- Before we get to explicit memory management, we need some notation for writing about memory
- A *pointer* is an address in memory, which means that it is a Large Meaningless Integer. You can examine its value, but it will only refer to the X-th memory location your program can access, which isn't very informative.
- The idea is to have the *value* of one memory location coincide with the *index* of another:
  - If location 2 contains a pointer, it points at the value 42 (in location 5)
  - If location 0 contains a pointer-to-pointer, it points at the value 5 (in location 2), which in turn points at the value 42 (in location 5), etc. etc.

Location	0	1	2	3	4	5...
Value	2	10	5	12	64	42...

Pointers in practice

- The concept itself isn't too bad, but manipulation can get tricky.
- Syntactically,

```
double value = 0.0;     // 'value' is a double-precision scalar
double *ptr1;           // 'ptr1' is a pointer to a double-prec. Scalar
double **ptr2;          // 'ptr2' is a pointer-to-pointer-to-double
ptr1 = &value;          // This gets the address of our 0.0-valued var.
ptr2 = &ptr1;           // This gets the address of its address
value = 3.14;           // Assign with 0 levels of indirection
*ptr1 = 2.71;           // Assign with 1 level of indirection
**ptr2 = 3.14;          // Assign with 2 levels of indirection
```

- All these assignments change the same memory – after *ptr1=2.71, printing 'value' will give you 2.71 also.
- Mind the lifespan of what you find the & of – if it's on stack, it will disappear with the block it lives in.

  (In particular – don't have functions return pointers to their own local values, those disappear at return)

Dynamic allocation

- Since we can now refer to memory, here's a dynamic allocation:

```
{
    char *mychars = (char *) malloc ( 128 * sizeof(char) );
    for ( int i=0; i<128; i++ )
        *(mychars + i) = i+64;
    free ( mychars );
}
```

- This dynamically gets a lump of 128 bytes (chars), sequentially assigns them values 64 to 192, and throws them away again.
- Not terribly useful in itself, **BUT** – the data given by 'malloc' isn't wiped out until it is explicitly removed with 'free', so the value of this pointer could be passed around as long as needed.
- Also, the amount of memory requested from malloc can depend on variables read from user, etc.
- Full flexibility under full responsibility: what you allocate should be freed, and freed exactly once.

NTNU
Norwegian University of Science and Technology

Arrays are pointers, pointers are arrays

- Looking a little closer at the assignment in

```
for ( int i=0; i<128; i++ )
    *(mychars + i) = i+64;
```

  the meaning is clearly *“assign i+64 to the value in location (mychars + i)”*, but it's not beautiful to look at.

- An equivalent way of writing the assignment

```
*(mychars + i) = i+64;
```

  is

```
mychars [ i ] = i + 64;
```

- This is just notation. Array indexing means “dereference the named pointer plus the index” either way.
- Declaring “char mychars[128];” globally will also make 'mychars' a pointer to 128 chars, they just won't be placed on the heap like 'malloc' does.

NTNU
Norwegian University of Science and Technology

Arrays and pointers II

- To define an array we need
  - Some space for several elements (allocated on stack or heap)
  - Knowledge of what type they are (C knows)
  - A pointer to the first element (What we just looked at)
  - A count of how many there are (Your problem)
- Pointers are all the same size, i.e. the length of a memory address. The reason we declare them with types is for C to handle what “ptr + 1” (i.e. “next element”) means:
  - If ptr points at chars, the next one is found at 1 byte higher
  - If ptr points at floats, the next one is found at 4 bytes higher
- The simplicity of this means there's no safeguard against accessing “my_array [65000]” even if you've only allocated 4 elements – the processor just adds an address and a multiple of the element size, and works on whatever it finds.
- *Following pointers where they shouldn't point will probably account for >90% of your program crashes.*

NTNU
Norwegian University of Science and Technology

The flexible mess of *

- In declarations, * means you are declaring a pointer to a type, instead of an instance of it.
- In arithmetic, * means multiplication. The difference is always clear to the compiler, and the careful programmer makes it clear from the source, too.
- In memory references, * means dereference, i.e. “follow the pointer and fill in the value it points at”.
- Taken together, this lets you write things like

  ```
  a *= *b * **c;
  ```

  in order to

  multiply (a) with (what b points at) times (what pointer-at-*c points at)

  or equivalently, to kill the maintenance engineer.
- Remember:
  - In declarations, * changes what the variable is
  - In expressions, * is something you do
  - Parentheses and whitespace are allowed

NTNU
Norwegian University of Science and Technology

Fundamental types

- “char” is a character (a byte, values def. in ASCII table)
- “int” is some sort of integer
- “long” is some kind of longer integer, “short” is a shorter one
- (“float” and “double” are IEEE-floating point numbers, but we won't need them)
- ints, longs, shorts etc. are horribly mis-defined in ANSI C (1989)
- ISO C99 rectifies a lot of it, but support isn't universal (Notable exception: MS Visual Studio)
- #include <stdint.h> gives exact types: int16_t, uint8_t, int32_t etc. are *exactly* 16, 8 and 32 bits, respectively
- C89 also mixes boolean and arithmetic expressions, accepting that “true” is equivalent to “not zero”
- In C99, including “<stdbool.h>” gives a 'bool' type, as well as keywords “true” and “false”. *(Old interpretation still holds, though)*
- “Strings” are just arrays-of-char, with 0 at the end

Type definitions

- typedef <type> mytype_t;

  lets you declare variables as “mytype_t”, giving them the type you specified in the typedef
- Along with the basic types, this is just aliases, but can clarify what the program means to a reader:

```
typedef int postal_code_t;
postal_code_t zipcode;        // this var. is clearly not going to be used
                              // for storing the height of mt. Kilimanjaro
```

- The real beauty is when the type-specifier defined is more complicated than just an int

Enumerated types

- typedef enum { A, B, O } bloodtype_t;
- This will give (irrelevant) magic integers to the names A, B and O
- bloodtype_t will still just be an alias for an int, but you get a little help for type safety: assigning something other than A, B or O to a bloodtype_t variable will produce a warning
- This is the method of choice for making your magic constants readable.
- You _can_ specify *enum {A=0, B=1, O=2}* and calculate with them too, but that defeats the point

NTNU
Norwegian University of Science and Technology

Structured types

- typedef struct { int x, y; } coords_t;

  defines a type from two integers named x and y – here, conveniently packaged to be 2D coordinates.
- coords_t myco = (coords_t) { .x = 100, .y = 200 };

  assigns (100,200) to the components.
- Elements are referred to with '.', as in

  printf (“coords (%d,%d)\n”, **myco.x, myco.y**);
- coords_t *myco = (coords_t *) malloc (sizeof(coords_t));

  gives a dynamically allocated pointer-to-coordinates
- (*myco).x = 60; will then assign the x-part. This is common enough that it has an equivalent notation

  myco->x = 60;

  for the sake of convenience

Self-referential structures

```
typedef struct elem {
    char *tag;
    struct elem *next;
} list_t;
```

- This defines a pointer-to-char, and a pointer to a next-element of the same kind – that is, this can be used to build linked lists of strings.
- We will make a lot of trees using this kind of structure.
- Note how the name given by the *typedef* can't be used before the typedef statement is complete, but the name of the struct-type itself can.

NTNU
Norwegian University of Science and Technology

Union types

typedef union { int i; float f; } int_or_float_t;

- This makes the declaration

  int_or_float_t myval;

  give space enough for an integer OR a float (based on whichever is longer).
- 'myval' can be either, but not both at the same time
- What it will be treated as depends on whether you write **myval.i** (now it's integer) or **myval.f** (now it's a float).
- These don't have a whole lot of applications in an age of large memory banks, but they are mentioned here because our parser-generator tool uses them in at least one capacity.

Function prototypes

- As indicated a while ago, if your program doesn't contain the definition of a function, the compiler still likes to know it's name, and the types of its arguments.
- That's so it can check that your function calls don't pass strings where doubles are required, and such.
- Prototypes are function declarations without the body, they specify return type, and a typed list of arguments, as in

  void my_function (int32_t a, int32_t b);

  → "my_function returns nothing, and takes 2 32-bit integers in".
- The names a,b are optional in the prototype, all that's needed is the *type signature*.

Getting clever with parameters

- void my_function (int32_t a, …);

  means that my_function takes *any number of parameters*, but at least one 32-bit integer.
- <stdargs.h> gives access to the rest (more on that shortly)
- *void my_function ( );*

  means the same thing as

  void my_function (…);

  for historical reasons. The proper way to write “my_function takes exactly zero arguments” is

  void my_function (void);

<stdarg.h> and va_list

```
void link_nodes ( int8_t nodes, … )
{
    va_list arglist;
    list_t *current, *new_node;
    va_start ( arglist, nodes );
    current = va_arg ( arglist, list_t * );
    for ( int8_t n=1; n<nodes; n++ )
    {
        new_node = va_arg ( arglist, list_t * );
        current->next = new_node;
        current = new_node;
    }
    va_end ( arglist );
    current->next = NULL;
}
```

- This illustrates the linking of a list_t list – analysis follows

NTNU
Norwegian University of Science and Technology

Getting clever with parameters II

- "va_list arglist" is a structure to track variable length argument lists. It needs to know where to start, so *va_start(arglist,nodes)* initializes it to start reading after the named argument ('nodes', which is the last named arg. in our prototype...)
- We also need to know (dynamically) how many arguments to read. Here, that is found by the integer "nodes", which is made to be a count of arguments to follow
- i.e. if we had list_t pointers a,b,c,d and wanted to link them, the call would be

  link_nodes (4, a, b, c, d);
- If there would be a 'list_t *e' also, this would handle it:

  link_nodes (5, a, b, c, d, e);

NTNU
Norwegian University of Science and Technology

Variable length argument lists, dissected

- The va_arg function updates the tracking structure, and requires the expected type of the next argument
- We have just looked straight into the heart of *printf* – reading the format string (“%d, %f, %x\n” and the like) permits counting how many more parameters to expect, and what types they will have
- Our function only expected *n* counts of the same type, so a total count variable is enough
- This is kind of an advanced topic, but we cover it because it's a smashing way to shorten the construction of large, nonuniform structures into a few lines of code.

Fun with the preprocessor

- We've covered types and memory management, which (apart from the ordinary statements) is mostly what C deals with
- In practice, the greater challenge of becoming effective with it is to put the preprocessor to good use
- At its best, it can make your program a short and readable thing where the preprocessor writes out all the repetitive and boring things for you
- At its worst, it can degenerate your program into a perfectly unreadable and incomprehensible blob of mysterious errors

#define = macro substitution

#define LIMIT 42

will make the p.proc. scan through the file and replace the string "LIMIT" with the string "42", before compilation.

(defs. to the preprocessor are called macros, and commonly uppercased when they affect program state)

- This is a slightly different way to name magic constants, with a slightly different use than enumerated types:
  - If you have the be-all and end-all of your magic numbers, enumerated types give you checking of expressions
  - If you want to compile a version of your program with LIMIT set to 84 instead, you can override its value without changing the program code

#ifdef / #ifndef / #endif – conditional compilation

```
#ifndef MYHEADER_H
#define MYHEADER_H
(...prototypes, rest of the file...)
#endif
```

- These make the preprocessor include what's between the clauses only if the requested macro is defined
- The mechanism above is a convention for header files which contain declarations which need to be read exactly once, to let them be #include-d from any number of files without causing repetition

#define with parameters

```
#define SUMSQ(x,y) ((x)*(x)+(y)*(y))
```

will expand the string

SUMSQ(4,5)

into the expression

((4)*(4)+(5)*(5))

before compilation.

- This permits writing shorthand for commonly used expressions without defining an entire function for them.
- Always parenthesize macro arguments – writing e.g.

```
#define F(x,y) (x*50+y)
```

will probably not give you what you want:

F(a-1,2) gives (a–1*50+2) which is (a-50+2),

as opposed to (a-1)*50+(2)

NTNU
Norwegian University of Science and Technology

A few bits and bobs which don't fit anywhere

- C99 support isn't on by default. GCC takes the flag '-std=c99' (or -std=gnu99 for an extended version) in order to enable it.
  - 'make CFLAGS=-std=c99 hello'

    or put “CFLAGS+=-std=c99” in a file called “Makefile” in your working directory
  - Most immediately, this allows you to write “for (int i=0; …)”, declarations in loop heads will cause compilation with older standards to abort....
- I haven't covered
  - Function pointers
  - Variadic macros
  - Labels & gotos (yikes!)
  - ...and a few other things – if the need arises, we'll take it in stride

NTNU
Norwegian University of Science and Technology

Wrapping up

- We have now zoomed through most of C by its principles, almost devoid of code and examples.
- The world is full of C code, reading and writing some is useful to get to grips with it.
- You can't *actually* learn C in 90 minutes, this was intended to cover the kind of material which is usually not covered in tutorials you can find online, etc.
- We don't *really* have the headroom to be a C course also, but it's possible to pick it up on the side if you put some effort into it.

  (it's been done before :))