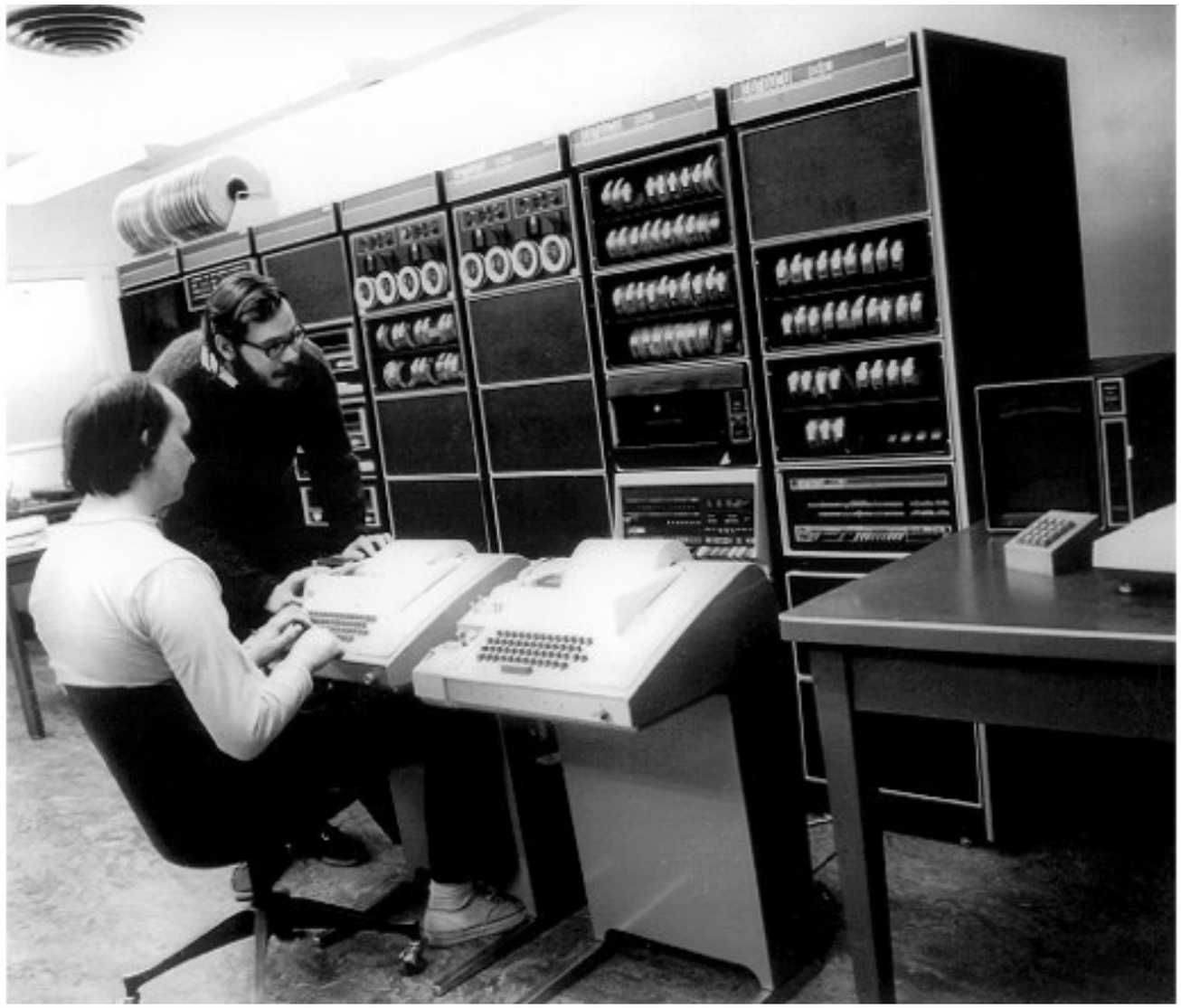
Functions

"to get a deeper understanding of the language"



Deep C - a 3 day course
Jon Jagger & Olve Maudal

exercise

- what does this say?

```
int get_value();
```

- try the following...

```
#include <stdio.h>

int get_value();

int main()
{
   printf("%d\n", get_value());
   printf("%d\n", get_value(42));
   printf("%d\n", get_value(42,24));
   return 0;
}

int get_value()
{
    return 42;
}
```

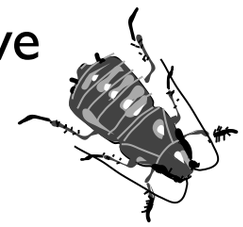
f() vs f(void)

if a function has no parameters say so explicitly with `void`

```
int rand();
```

old-style function declaration

provides no parameter information;
the definition of rand can have
any number of parameters!



```
int rand(void);
```

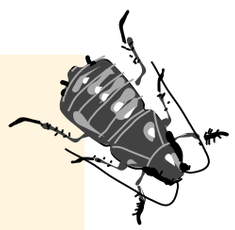
new-style function prototype

rand has no parameters;
the definition of rand must have
no parameters

Parameter-argument number and type mismatches are **not** caught with the -Wall option. Read -Wall as -Wmost!

call.c

```
int func();

int main(void)
{
    return func(42);
}
```

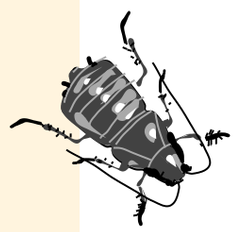


```
$ gcc -Wall call.c
$
```

Using -Wstrict-prototypes forbids old style parameterless function declarations

call.c

```
int func();

int main(void)
{
    return func(42);
}
```

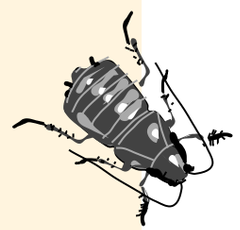


```
$ gcc ... -Werror -Wstrict-prototypes call.c
error: function declaration isn't a prototype
$
```

Using -Wstrict-prototypes ensures modern function prototypes are used and catches argument-parameter mismatches

call.c

```
int func(void);

int main(void)
{
    return func(42);
}
```



```
$ gcc ... -Werror -Wstrict-prototypes call.c
error: too many arguments to function 'func'
$
```

inline functions

- there must be a definition in the translation unit
- does not affect sequence point model – there is still a sequence point before a call to an inline function
- prefer inlining over macros

c99

is_even.h

```
#ifndef IS_EVEN_INCLUDED
#define IS_EVEN_INCLUDED

#include <stdbool.h>

static inline bool is_even(int value)
{
    return value % 2 == 0;
}

#endif
```

pass by pointer

- use a pointer to a non-const
- if the definition needs to change the target

delay.h

```
void delay( struct date * when);
```

the lack of a `const` here means
`delay` might change `*when`

```
struct date * when = &due;
```

```
#include "delay.h"

int main(void)
{
    struct date due = { 2012, march, 28 };
    delay(&due);
    ...
}
```

pass by value

changing the parameter does not change the argument

```
bool search(
    const int values[], size_t from, size_t to,
    int find)
{
    while (from != to && values[from] != find)
    {
        from++;
    }
    return from != to;
}
```

```
int main(void)
{
    ...
    ... search(array, 0, size, 42);
    ...
}
```

```
size_t from = 0;
```

pass by value

- works for enums and structs too
- but not for arrays

date.h

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

const char * day_name(struct date when);
```

```
#include "date.h"

int main(void)
{
    struct date today = { 2012, march, 28 };
    ...
    puts(day_name(today));

    assert(today.year == 2012);
    assert(today.month == march);
    assert(today.day == 28);
}
```

```
Wednesday
```

pass by pointer to const

- often an efficient alternative to pass by copy
- except that the parameter can be null

date.h

```
const char * day_name(const struct date * when);
```

the `const` here promises that `day_name` wont change `*when`

```
#include "date.h"

int main(void)
{
    struct date today = { 2012, march, 28 };
    ...
    puts(day_name(&today));

    assert(today.year == 2012);
    assert(today.month == march);
    assert(today.day == 28);
}
```

```
Wednesday
```

parameter order

- list output parameters first
- loosely mimics assignment

```
char * strcpy(char * dst, const char * src);
```

```
#include <string.h>
...
{
    const char * from = “Hello”;
    char to[128];
    ...
    //      to = from
    strcpy(to , from);
}
```

you can list the type and its qualifiers in either order

```
const char * day_name(const struct date * at);
```

```
const char * day_name(struct date const * at);
```

The second style (const last) is common in C++ but not C.
The rationale for prefering the second style is that it allows you to read a declaration from right to left:

"at is a pointer to a const date"

register variables

- a speed optimization hint to the compiler
- compiler will use registers as best it can anyway
- effect is implementation defined
- register variables can't have their address taken
- don't use!

```
void send(register short * to,
          register short * from,
          register int count)
{
    register int n = (count + 7) / 8;
    switch (count % 8)
    {
    case 0 : do {  *to++ = *from++;
    case 7 :       *to++ = *from++;
    case 6 :       *to++ = *from++;
    case 5 :       *to++ = *from++;
    case 4 :       *to++ = *from++;
    case 3 :       *to++ = *from++;
    case 2 :       *to++ = *from++;
    case 1 :       *to++ = *from++;
                } while (--n > 0);
    }
}
```

??

local statics

- a local variable can have static storage class
- a local variable with 'infinite' lifetime
- best avoided – subtle and hurts thread safety
- but ok for naming magic numbers (as are enums)

```
int remembers(void)
{
    static int count = 0;
    return ++count;
}
```

```
void send(short * to, short * from, int count)
{
    static const int unrolled = 8;

    int n = (count + unrolled - 1) / unrolled;
    switch (count % unrolled)
    {
        ...
    }
}
```

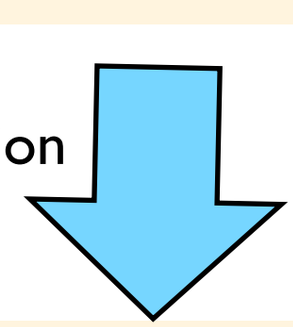
__func__

- the name of the current function is available
- via the reserved identifier __func__
- use for logging

```
void some_function(void)
{
    puts(__func__);
}
```

c99

as-if compiler translation



```
void some_function(void)
{
    static const char __func__[] =
        "some_function";
    puts(__func__);
}
```

... variadic functions

- functions with a variable no. of arguments
- helpers in <stdarg.h> provide type-unsafe access
- be careful ... arguments are *not* default promoted

```
#include <stdarg.h>

int my_printf(const char * format, ...)
{
  va_list args;
  va_start(args, format);
  for (size_t at = 0; format[at] != '\0'; at++)
  {
    switch (format[at])
    {
      case 'd': case 'i':
        print_int   (va_arg(args, int   )); break;
      case 'f': case 'F':
        print_double(va_arg(args, double)); break;
      ...
    }
  }
  va_end(args);
}
```

function pointers

- `( )` is a binary operator with very high precedence
- `f(a,b)` is like an infix version of `( )( f, a, b )`
- you can name a function without calling it!
- the result is a strongly typed function pointer

* is needed here

```
#include <stdio.h>

int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }

int main(int argc, char * argv[])
{
    int (*f)(int,int) =
        argc % 2 == 0 ? add : sub;

    printf("%d\n", f(3, 1));
}
```

function pointer arguments

- function pointers can be function parameters!
- * is optional on the parameter

* is not needed here

```
int call(int (*f)(int,int))
{
    return (*f)(3, 1);
}
```

equivalent

```
int call(int f(int,int))
{
    return f(3, 1);
}
```

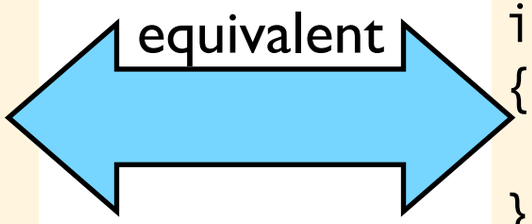
```
#include <stdio.h>

int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }

int main(int argc, char * argv[])
{
    int (*f)(int,int) =
        argc % 2 == 0 ? add : sub;

    printf("%d\n", call(f));
}
```

function pointer arguments

- `typedef` can often help

```
typedef int func(int, int);
```

```
int call(func * f)
{
    return (*f)(3, 1);
}
```




```
int call(func f)
{
    return f(3, 1);
}
```

summary

don't use auto or register keywords
don't use static local variables unless const
don't use f( ); declarations
do use f(void); prototypes (-Wstrict-prototypes)

do pass by copy for built in types and enum...
- they are small and they will stay small
- copying is supported at a low level, very fast
- sometimes for structs as a no-alias no-indirection optimization

do pass by plain pointer...
- when the function needs to change the argument

do pass by pointer to const (mimic pass by copy)
- for most structs
- they are not small and they only get bigger!
- very fast to pass, but be aware of cost of indirection