The C Programming language

(continued)

Functions

Last slides from the previous week

hello.c

```
#include "out.h"

int main(void)
{
    printme();
    return 0;
}
```

out.c

```
#include <stdio.h>

void printme()
{
    printf("Hello 2140!\n");
}
```

Makefile

```
hello: hello.c out.c
	gcc -o $@ $^

hello: out.h
```

out.h

```
#ifndef OUT_H
#define OUT_H

void printme();

#endif
```

Functions

```c
#include <stdio.h>

void printtext();
void printreturn();

int main(void)
{
    printtext();
    return 0;
}

void printtext()
{
    printf("Hello 2140!");
    printreturn();
}
```

Recursion:

- frequent in algorithms
- rarely used in Python and Java
- very frequent in Haskell and other functional languages

Look at a recursive example:

It works as well. Until it doesn't.

Any idea what's wrong?

- Every running process has at least one area called a **"stack"**.
- Most simple variable in function use some space from the **stack**.
- Also, when you call a function, it uses some space from the **stack**.
- **Eventually, you run out!**

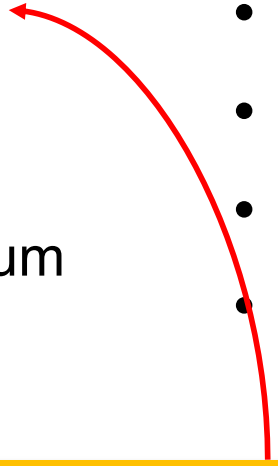
Data types in C

C has a lot of data types:

- basic
  - 10 integer types
  - 3 floating point types
  - void
- derived
  - array, struct, union, enum
  - pointer types

C does not have:

- boolean
- complex numbers
- classes
- reference types
- weak pointers



It has so many because **you** need to know:

- how many bytes each one has
- what each bit means

Data types in C

C has several different integer types.
Their first difference is their size in memory.

Name	Number of bytes	Actual size on login.ifi.uio.no
`char`	`1`	`1`
`short`	`>= 2, <= sizeof(int)`	`2`
`int`	`>= 2`	`4`
`long`	`>= 4, >= sizeof(int)`	`8`
`long long`	`>= sizeof(long)`	`8`

Printing integer types

```c
#include <stdio.h>

int main(void)
{
    int i = 42;
    printf("i=%d\n",i);
    return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
    long i = 42;
    printf("i=%ld\n",i);
    return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
    long long i = 42;
    printf("i=%lld\n",i);
    return 0;
}
```

Printing integer types

```c
#include <stdio.h>

int main(void)
{
    short i = 42;
    printf("i=%d\n",i);
    return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
    char i = 42;
    printf("i=%d\n",i);
    return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
    char i = 42;
    printf("i=%c\n",i);
    return 0;
}
```

Types in C: signed / unsigned

- Integer types exist as signed or unsigned
  e.g. `signed char`, `unsigned char`

- They have the same size in memory, but different number ranges

  - `unsigned char`
  - (numbers 0 .. 255)

binary 00000000 = decimal 0
binary 01111111 = decimal 127
binary 10000000 = decimal 128
binary 11111110 = decimal 254
binary 11111111 = decimal 255

  - (`signed`) `char`
  - (numbers -128 .. 127)

binary 00000000 = decimal 0
binary 01111111 = decimal 127
binary 10000000 = decimal -128
binary 11111110 = decimal -2
binary 11111111 = decimal -1

Printing integer types

```c
#include <stdio.h>

int main(void)
{
    unsigned char      a = 1;
    unsigned short     b = 1;
    unsigned int       c = 1;
    unsigned long      d = 1;
    unsigned long long e = 1;

    printf("a=%u b=%u c=%u d=%lu e=%llu\n", a,b,c,d,e);
    return 0;
}
```

Data types in C

Floating point types

Name	Number of bytes	Actual size on login.ifi.uio.no
`float`	`>= 4`	4
`double`	`>= 8, > sizeof(float)`	8
`long double`	`>= sizeof(double)`	16

"Nothing" data type

Name	Number of bytes	Actual size on login.ifi.uio.no
`void`	0	0

Printing floating point types

```c
#include <stdio.h>

int main(void)
{
    float       a = 1.0;
    double      b = 1.0;
    long double c = 1.0;

    printf("a=%f b=%lf c=%llf\n", a,b,c);
    printf("a=%g b=%lg c=%llg\n", a,b,c);
    printf("a=%e b=%le c=%lle\n", a,b,c);
    return 0;
}
```

Arithmetic operators

+	add	++	post-increment
-	subtract		pre-increment
*	multiply	--	post-decrement
/	divide		pre-decrement
%	modulo		

```
int a;
a=1; a++; // a: 2
a=1; a--; // a: 0
a=1; ++a; // a: 2
a=1; --a; // a: 0
```

```
int a,b;
a=1; b=a++; // b: 1
a=1; b=a--; // b: 1
a=1; b=++a; // b: 2
a=1; b=--a; // b: 0
```

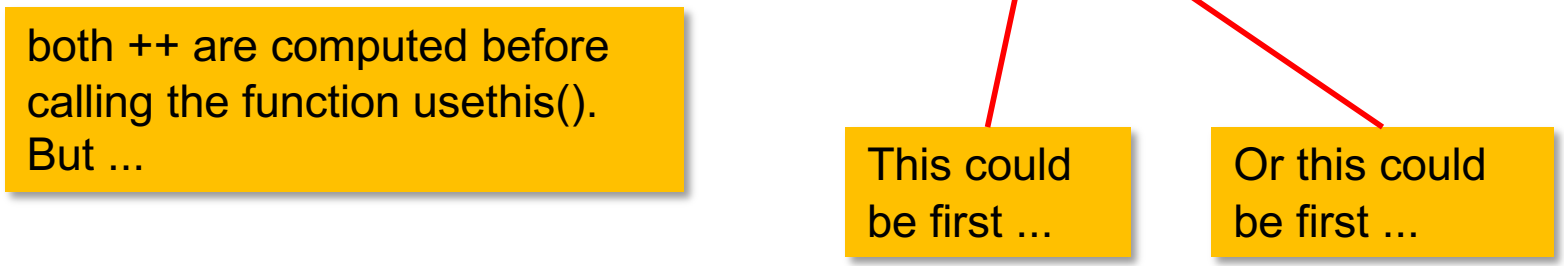
Arithmetic operators

```
int a,b;
a=1; b=a++; // b: 1
a=1; b=a--; // b: 1
a=1; b=++a; // b: 0
a=1; b=--a; // b: 2
```

Increment operators keep code short. Use them.
But they can be used for really confusing code.

The C11 standard says that the following is **undefined**

```
b = usethis(++a,++a);
```



Assignment operators

=	set
+=	add and set
-=	subtract and set
*=	multiply and set
/=	divide and set

```
int a;
a=5; a += 5;  // a: 10
a=5; a *= a;  // a: 25
a=5; a *= 10; // a: 50
```

Comparison operators

==	equal
!=	unequal
<	Less than
<=	Less than or equal
>	Larger than
>=	Larger than or equal

== and = are very different

Logical operators

Logic values

There is no boolean type in C.

false: 0

true: **absolutely every other value**, even -1

Logical operators

&& || !

```
int b, a = 10000;
b = a;   // b: 10000
b = !a;  // b: 0
b = !!a; // b: 1
```

Binary operators

Binary operators work on every bit of an integer type

~	not
&	and
|	or
^	Xor
<<	Left shift
>>	Right shift

a	b	~a	a & b	a | b	a ^ b
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Be very careful when you mix logical and binary operators.

```
int a = 1 && 4;   // a = 1
int b = 1 & 4;    // b = 0

// 1: 0001
// 4: 0100
```

Working with bits

```c
#include <stdio.h>

int main(void)
{
    unsigned int a = 1;
    unsigned int b = 2;
    unsigned int c = 4;
    unsigned int d = 16;

    printf("a=%x b=%x c=%x d=%x\n", a,b,c,d);

    a = a | b | c;
    printf("a=%x b=%x c=%x\n", a,b,c);

    a = a & ~b;
    printf("a=%x b=%x c=%x\n", a,b,c);
    return 0;
}
```

A modern Linux's printf has also %b to print the bits for you.
IFI's machines are not modern.

Quick look at `char`

```c
#include <stdio.h>

int main(void)
{
    char c;

    c = 42;         printf("c: %d %x %c\n", c, c, c);
    c = 'b';        printf("c: %d %x %c\n", c, c, c);
    c = 'b' - 'a'; printf("c: %d %x %c\n", c, c, c);
    c = '0' + 7;    printf("c: %d %x %c\n", c, c, c);

    int x = 12;
    int y = 12 % 10;
    int z = 12 / 10;
    printf("%c %c\n", '0'+z, '0'+x);

    return 0;
}
```

Quick look at `char`

```c
#include <stdio.h>

int main(void)
{
    unsigned int bits = 0x01 | 0x02 | 0x08 | 0x40;

    printf("0x");
    printf("%c", '0' + !!(bits & 0x01));
    printf("%c", '0' + !!(bits & 0x02));
    printf("%c", '0' + !!(bits & 0x04));
    printf("%c", '0' + !!(bits & 0x08));
    printf("%c", '0' + !!(bits & 0x10));
    printf("%c", '0' + !!(bits & 0x20));
    printf("%c", '0' + !!(bits & 0x40));
    printf("%c", '0' + !!(bits & 0x80));
    printf("\n");

    return 0;
}
```

The result of 0x04 & 0x04 is 0x04.
The two not-signs turn it into 1.

You could of course also use %d here, but this idea works better when you want to turn a bitfield into a C-string.
But that makes sense only when you know structs and arrays in C.

More operators

Very important "other" operators

`&`	reference
`*`	dereference
`->`	access
`.`	access
`[ ]`	array
`( )`	call parameters
`? :`	compact if then else
`,`	chain operations
`sizeof`	byte size of operand

```
int a;

printf("size: %ld\n", sizeof(int));

printf("size: %ld\n", sizeof(a));

printf("size: %ld\n", sizeof a);

printf("size: %ld\n", sizeof int);
```

More operators

Very important "other" operators

`&`	reference
`*`	dereference
`->`	access
`.`	access
`[ ]`	array
`( )`	call parameters
`? :`	compact if then else
`,`	chain operations
`sizeof`	byte size of operand

new since C11
for
- macro writers
- writers of high-performance libraries

`_Alignof`	alignment operator
`_Generic`	generic selection
`_Noreturn`	function specifier
`_Static_assert`	static assertion

Syntax challenges

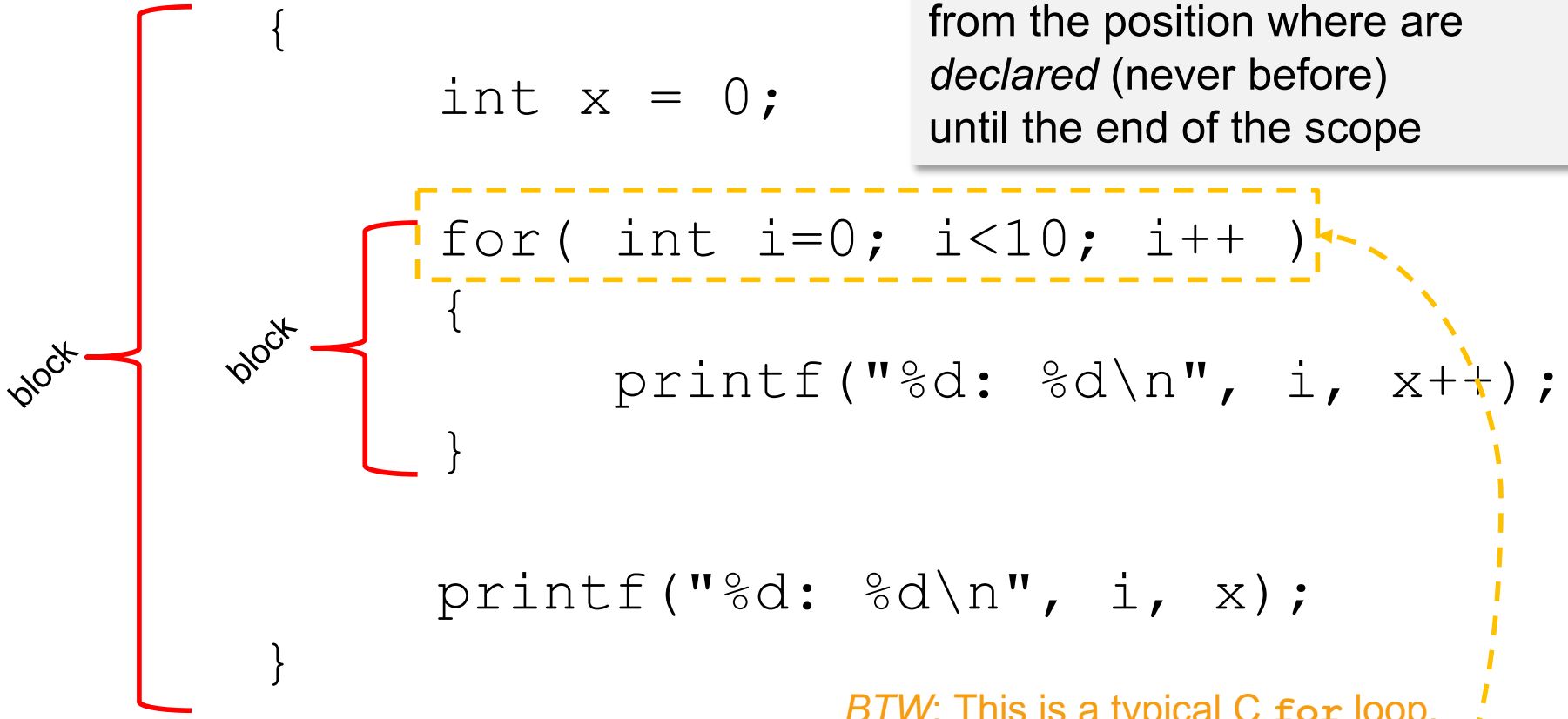
```
int x, y, z;
x = 10; y = 10; z = 10;

x = y = z == 10;

x = y == (z = 10);
```

Blocks

Blocks define **scope.**
Things declared in a scope exist
from the position where are *declared* (never before)
until the end of the scope




```
int main(void)
{
        int x = 0;

    for( int i=0; i<10; i++ )
    {
            printf("%d: %d\n", i, x++);
    }

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

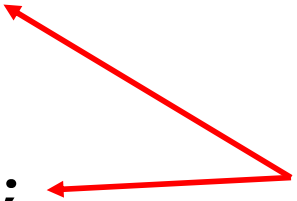
BTW: This is a typical C `for` loop.
C cannot do range-based for loops. C knows nothing about ranges.
We will get back to this.

Blocks

```
int main(void)
{
      int x = 1;
      printf("%d\n",x);

      {
            x = 2;
            printf("%d\n",x);
      }

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



Guess:
What is the output?

Blocks

```
int main(void)
{
      int x = 1;
      printf("%d\n",x);

      {
            int x = 2;
            printf("%d\n",x);
      }

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

Guess:
Is this a problem for C?
Do you get a warning?

Guess:
What is the output?

Control flow statements

`while(EXPR) BODY`	while loop, condition checked before first enter
`do BODY while(EXPR);`	while loop, condition not checked before first enter
`for( EXPR; EXPR; EXPR ) BODY`	for loop
`if(EXPR) BODY`	if-statement
`if(EXPR) BODY else BODY`	if-else-statement

`switch(EXPR) {`
`case VALUE: BODY`
`case VALUE: BODY`
`...`
`default: BODY }`
	multi-case statement,
	VALUE can only be an integer number, or
	a macro that becomes an integer before compilation, or
	an enumerator (`enum`) entry, which is the compiler turns into an integer number

`goto LABEL:`	jump inside a function
`return EXPR`	returning from a function
`break;`	continue after the innermost switch, for, while or do-while block
`continue;`	jump straight to the innermost for, while or do-while expression (EXPR)

`for` and `while`

```
int main(void)
{
    int i;

    for( i=0; i<10; i++ )
    {
        printf("%d\n", i);
    }
    // i is still here
}
```

```
int main(void)
{
    int i;

    i=0;
    while( i<10 )
    {
        printf("%d\n", i);
        x++;
    }
    // i is still here
}
```

old-fashioned variable declaration at the start of the block

`for` is just a quicker way of writing a `while` loop

`for` and `while`

```c
int main(void)
{
    for( int i=0; i<10; i++ )
    {
        printf("%d\n", i);
    }
    // i is gone
}
```

```c
int main(void)
{
    {
        int i = 0;
        while( i<10 )
        {
            printf("%d\n", i);
            x++;
        }
        // i is still here
    }
    // i is gone
}
```

modern variable declaration in the for expression
`for` replaces a `while` loop inside a block with declaring the loop variables

Scoping rules to save memory

```
#include "printme.h"

int main(void)
{
  {
    unsigned long long a[10000];
    for(int i=0; i<10000; i++)
      a[i] = i;
    for(int i=1; i<10000; i++)
      a[i-1] += a[i];
    printme(a);
  }
  {
    double b[10000];
    for(int i=0; i<10000; i++)
      b[i] = 1.0/i;
    for(int i=1; i<10000; i++)
      b[i-1] *= b[i];
    printyou(b);
  }
}
```



This is a new thing:

an **array**

This one consists of 10000 unsigned long longs, so it takes 80000 bytes of memory.

Scoping rules to save memory

```
#include "printme.h"

int main(void)
{
  {
    unsigned long long a[10000];
    for(int i=0; i<10000; i++)
      a[i] = i;
    for(int i=1; i<10000; i++)
      a[i-1] += a[i];
    printme(a);
  }
  {
    double b[10000];
    for(int i=0; i<10000; i++)
      b[i] = 1.0/i;
    for(int i=1; i<10000; i++)
      b[i-1] *= b[i];
    printyou(b);
  }
}
```

```
#include <stdio.h>

void printme(unsigned long long v[])
{
  for(int i=0; i<10000; i++)
    printf("%lld ",v[i]);
  printf("\n");
}

void printyou(double v[])
{
  for(int i=0; i<10000; i++)
    printf("%lf ",v[i]);
  printf("\n");
}
```

The programmer promises the compiler that `a` is not needed after the first block. OK to reuse the space for `b`. **It is your choice:**
The compiler has no way of knowing whether `printme` keeps the **address** of `a` for use in `printyou` or not.

Arrays

```c
#include <stdio.h>

void func( int z[] )
{
    printf("sizeof(z): %ld\n", sizeof(z));
}

int main(void)
{
    int x[10] = { 0, 1, 2, 3, 4 };
    int y[]   = { 0, 1, 2, 3, 4 };
    printf("sizeof(x): %ld\n", sizeof(x));
    printf("sizeof(y): %ld\n", sizeof(y));
    func(x);
    func(y);
    return 0;
}
```

Compiler has no idea how many there are.
Compiler uses a memory address instead.

10 4-byte integers as you instructed.

5 4-byte integer because you gave it exactly 5

Output:

```
sizeof(x): 40
sizeof(y): 20
sizeof(z): 8
sizeof(z): 8
```

IFI compiler warns:

```
"warning: 'sizeof' on array function parameter 'z' will return size of 'int *'"
```

Many compilers don't warn about this.

Arrays

C-strings are arrays of char that end with an invisible 0-byte

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

int main(void)
{
    char s[] = "four\n";

    printf(s);
    printf("sizeof %d\n", sizeof(s));
    printf("length %d\n", strlen(s));
    printf("length %c\n", s[2]);
    return 0;
}
```

Arrays

C-strings are arrays of char that end with an invisible 0-byte

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

int main(void)
{
    char b;

    {
        b = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
            "ABCDEFGHIJKLMNOPQRSTUVWXYZ"[26];
    }

    printf("%c\n",b);

    return 0;
}
```

We can access a character in a C-string directly.

This example is extreme because we have no way of accessing the C-string again. Even the block where it was defined is concluded.

The array nature of the string becomes really obvious.

Addresses

Everything that has a size does also have an address. We can print these addresses.

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

int x;
int main(void)
{
    char c;
    char s[] = "string";

    printf("address of x: %lx\n", &x);
    printf("address of c: %lx\n", &c);
    printf("address of s: %lx\n", &s[0]);
    printf("address of s: %lx\n", s);
    return 0;
}
```

address of a global variable

address of a local variable

address of the **first byte of** the C-string

shorthand for &s[0]

An `int` on our machines is 4 bytes long. Of course, each byte has its own address. `&x` gives us the numerically lowest memory address of the 4.

Addresses

Everything that has a size does also have an address. We can print these addresses.

```
#include <stdio.h>

int main()
{
    char c;

    *&c = 'a';

    printf("%c %c %lx\n",c,*&c,&c);
    return 0;
}
```

an uninitialized char

Take the address `&c` of `c`.
Then use `*` to dereference the address to get the space at the address.

The `*` operator has no idea that there is something named `c`.

But the compiler *infers* that it should be `char` because `c` was a `char`.

print `c`
It does now contain `'a'`

Addresses

A pointer is a variable that can contain an address. On a 64bit computer its size is always 8 bytes.

```
#include <stdio.h>

int main()
{
    char c;

    char* p = &c;

    *p = 'a';

    printf("%c %c %lx\n",c,*p,p);
    return 0;
}
```

We can store the address in a variable whose type is `char*`. We call this a "char pointer".

And we dereference it to write into the place in memory that is identified by the address stored in `p`. Just like before.

address arithmetics

```c
#include <stdio.h>

int main() {
    int a[] = { 1,2,3,4,5,0 };
    for( int i=0; a[i] != 0; i++ ) {
        printf("*%lx = %d\n", &a[i], a[i]);
    }
    return 0;
}
```

This can be abbreviated:
`int p=a;`

```c
#include <stdio.h>

int main() {
    int a[] = { 1,2,3,4,5,0 };
    for( int* p=&a[0]; *p!=0; p++ ) {
        printf("*%lx = %d\n ", p, *p);
    }
    return 0;
}
```

address arithmetics

```
#include <stdio.h>

int main() {
    int a[] = { 1,2,3,4,5,0 };
    for( int i=0; a[i] != 0; i++ ) {
```

This can be abbreviated:
`int p=a;`

There was no lecture time for more.

But I really recommend that you compile the program on slides 44/45 to see scoping in action.
First, try the code from slide 44:

```
gcc -o program sourcefile.c
```

Second, to the same with code from slide 45.
Third, time, try:

```
gcc -O3 -o program sourcefile.c
```

address arithmetics

```c
#include <stdio.h>

int main() {
    char a[] = "Hallo!";
    for( int i=0; a[i] != 0; i++ ) {
        printf("*%lx = %c\n", &a[i], a[i]);
    }
    return 0;
}
```

```c
#include <stdio.h>

int main() {
    char a[] = "Hallo!";
    for( char* p=&a[0]; *p!=0; p++ ) {
        printf("*%lx = %c\n ", p, *p);
    }
    return 0;
}
```

```c
#include <stdio.h>

int main() {
    char* p = "Hallo!";
    for( ; *p!=0; p++ ) {
        printf("*%lx = %c\n ", p, *p);
    }
    return 0;
}
```

address arithmetics

```c
#include <stdio.h>

int main()
{
    char a[] = "Hallo!";

    char* fwd = &a[0];
    char* bak = &a[0];

    while(*bak) bak++;

    while( fwd <= bak )
    {
        bak--;
        printf("%c %c\n", *fwd, *bak);
        fwd++;
    }
    return 0;
}
```

address arithmetics

Walking through a C-string

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

int main(void)
{
    char* b = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
              "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    b = b + 26;
    for( int i=0; i<26; i++ )
    {
        printf("%c%c ", *(b+i), *(b-i-1));
    }
    printf("\n");
}
```

address arithmetics

Walking t

A historical leftover from K&R:
`char * a, c;`
means that `a` is a `char*` (char pointer), while `c` is a `char`.
I think that is utterly confusing and I declare every pointer separately.

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

int main(void)
{
    char* b = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
              "ABCDEFGHIJKLMNOPQRSTUVWXYZ", *c;

    for( c=(b+=26)-1; *b != 0 ; b++, c-- )
    {
        printf("%c%c ", *b, *c );
    }
    printf("\n");
}
```

Another piece of unreadable crazyness.
You can write this, but debugging a piece of code like this is nearly impossible.

Scoping rules to save memory

```c
#include "printme.h"

int main(void)
{
    int a[100];
    for(int i=0; i<100; i++)
      a[i] = i;
    keep(a);

    float b[100];
    for(int i=0; i<100; i++)
      b[i] = 1.0/i;
    print(b);
}
```

```c
#include <stdio.h>

static int* q = 0;

void keep(int* a)
{
  q = a;
}

void print(float* v)
{
  for(int i=0; i<100; i++)
    printf("%d: %lf ", q[i], v[i]);
  printf("\n\n");
}
```

Use it later?
Sure.

A variable in global memory.
Invisible to other modules because it is `static`.

Storing the address of an array of unknown lifetime that comes from a different module for later?
No problem in C.

Scoping rules to save memory

```
#include "printme.h"

int main(void)
{
  {
    int a[100];
    for(int i=0; i<100; i++)
      a[i] = i;
    keep(a);
  }
  {
    float b[100];
    for(int i=0; i<100; i++)
      b[i] = 1.0/i;
    print(b);
  }
}
```

Now add scopes

```
#include <stdio.h>

static int* q = 0;

void keep(int* a)
{
  q = a;
}

void print(float* v)
{
  for(int i=0; i<100; i++)
    printf("%d: %lf ", q[i], v[i]);
  printf("\n\n");
}
```

Use it later? Oops.

With simple code like this, a still probably still around. But you have given the compiler **permission to reuse** its memory. Encourage reuse with –O3.

Swapping example

```
#include <stdio.h>

void swap(int a, int b)
{
    int c=a; a=b; b=c;
}
void newSwap(int *a, int *b)
{
    int c=*a; *a=*b; *b=*c;
}

int main()
{
    int u = 1;
    int v = 100;
    printf("%d %d\n", u, v);
    swap(u, v);       printf("%d %d\n", u, v);
    newSwap(&u, &v); printf("%d %d\n", u, v);
    return 0;
}
```

Interpret memory how you choose

```c
#include <stdio.h>

int main()
{
    char array[] = { 8, 7, 6, 5, 4, 3, 2, 1 };

    unsigned long long a = ((unsigned long long*)array)[0];
    printf("%llx %llu\n", a, a);

    unsigned int b = ((unsigned int*)array)[0];
    printf("%x %u\n", b, b);

    unsigned short c = ((unsigned short*)array)[0];
    printf("%x %u\n", c, c);

    unsigned long long d = (unsigned long long)array[0];
    printf("%llx %llu\n", d, d);

    return 0;
}
```

Stack variables

```
#include <stdio.h>

void arrayfun( int sz )
{
    int array[sz];
    printf("Array size %d: ", sz);
    for( int i=0; i<sz; i++ )
        array[i] = i;
    printf(" last=%d\n", array[sz-1]);
}

int main()
{
    for( int i = 1; i!=0; i <<= 1 )
    {
         arrayfun( i );
    }
    return 0;
}
```

Dynamic memory

```
#include <stdio.h>

void arrayfun( int sz )
{
    int* array = malloc( sz * sizeof(int) );
    printf("Array size %d: ", sz);
    for( int i=0; i<sz; i++ )
        array[i] = i;
    printf(" last=%d\n", array[sz-1]);
    free(array);
}

int main()
{
    for( int i = 1; i!=0; i <<= 1 )
    {
        arrayfun( i );
    }
    return 0;
}
```

There is more in the C language

- struct
- union
- enum
- bit fields
- bit shifting
- ...