Pointers in C

Hayo Thielecke
University of Birmingham
http://www.cs.bham.ac.uk/~hxt

February 9, 2016

C, C++, and Java

C

basic imperative language (assignment, while, functions, recursion)

+ malloc and free; no garbage collector

+ pointers combined with other language features

C++

basic imperative language (assignment, while, functions, recursion)

+ new and delete; no garbage collector

+ object orientation

+ templates

Java

basic imperative language (assignment, while, functions, recursion)

+ simplified version of C++ object-orientation

+ garbage collector $\Rightarrow$ programmer can be naive about memory

Why C is (still) important

- Bits have not gone out of fashion (though there are more of them)
- systems programming
- "portable assembly language"
- $\Rightarrow$ prerequisite for OS module
- compilers: Clang is written in C++
- security: buffer overflow $\Rightarrow$ catastrophic failure
  see also Heartbleed bug
- extensions of C, e.g. CUDA C, OpenCL for programming graphics processors
- different view of programming from higher level languages

Factorial in C

```
int factorial(int n)
{
  if(n == 0)
    return 1;
  else
    return n * factorial(n - 1);
}
```

A function in C is like a method in Java without any objects.
More or less like `public static`.

Factorial in C without recursion

```
int factorial2(int n)
{
    int res = 1;
    while(n > 0) res *= n--;
    return res;
}
```

How C is unlike Java

Here is some Linux kernel code [1]

```
int (*open)(struct device *dev);
int (*stop)(struct device *dev);
int (*hard_start_xmit) (struct sk_buff *skb,
                        struct device *dev);
```

Does that look like Java?

Well, there is `int`. And semicolons. ☺

Pointers to structures and functions, ∗.

[1] http://www.tldp.org/LDP/tlk/ds/ds.html

How C is unlike Java

Here is some Linux kernel code [1]

```
int (*open)(struct device *dev);
int (*stop)(struct device *dev);
int (*hard_start_xmit) (struct sk_buff *skb,
                        struct device *dev);
```

Does that look like Java?
Well, there is `int`. And semicolons. ☺
Pointers to structures and functions, $*$.

[1] `http://www.tldp.org/LDP/tlk/ds/ds.html`

How C is unlike Java

Here is some Linux kernel code [1]

```
int (*open)(struct device *dev);
int (*stop)(struct device *dev);
int (*hard_start_xmit) (struct sk_buff *skb,
                        struct device *dev);
```

Does that look like Java?
Well, there is `int`. And semicolons. ☺
Pointers to structures and functions, `*`.

[1] http://www.tldp.org/LDP/tlk/ds/ds.html

Outline of pointers in C part of the module

Pointers are the fundamental new feature of C compared to the languages you have been taught previously.
Pointers are everywhere in C:

1. pointer types and operations
2. pointer equality
3. malloc and free
4. structures and pointers
5. pointer arithmetic
6. strings and pointers
7. function pointers

Syntax:

```
* & = == malloc free struct . -> ++ -- (*f)()
```

Back to basics: what is the meaning of =

What is the meaning of:

```
x = x + 1;
```

Does it mean: revolutionary new result in algebra:
Like, every number is equal to the next biggest number?

$$2 = 2 + 1$$

No so much.
Basic imperative programming: abstraction of the memory as named boxes that contain values.

Back to basics: what is the meaning of =

What is the meaning of:

```
x = x + 1;
```

Does it mean: revolutionary new result in algebra:
Like, every number is equal to the next biggest number?

$$2 = 2 + 1$$

No so much.
Basic imperative programming: abstraction of the memory as named boxes that contain values.

L and R values in C

What is the meaning of:

```
x = x + 1;
```

before:

x $\boxed{2}$

after:

x $\boxed{3}$

The x on the left of the = refers to the address (L-value) of x.
The x on the right of the = refers to the contents (R-value) of x.
In C, L-values are particularly important, in the form of pointers.

p

(Haskell is the opposite extreme.)

L and R values in C

What is the meaning of:

```
x = x + 1;
```

before:

x $\boxed{2}$

after:

x $\boxed{3}$

The `x` on the left of the = refers to the address (L-value) of `x`.

The `x` on the right of the = refers to the contents (R-value) of `x`.

In C, L-values are particularly important, in the form of pointers.

(Haskell is the opposite extreme.)

L and R values in C

What is the meaning of:

```
x = x + 1;
```

before:
`x` $\boxed{2}$

after:
`x` $\boxed{3}$

The `x` on the left of the = refers to the address (L-value) of `x`.
The `x` on the right of the = refers to the contents (R-value) of `x`.
In C, L-values are particularly important, in the form of pointers.

(Haskell is the opposite extreme.)

L and R values in C

What is the meaning of:

```
x = x + 1;
```

before:

x $\boxed{2}$

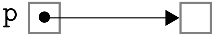
after:

x $\boxed{3}$

The `x` on the left of the = refers to the address (L-value) of `x`.

The `x` on the right of the = refers to the contents (R-value) of `x`.

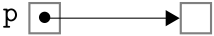
In C, L-values are particularly important, in the form of pointers.



(Haskell is the opposite extreme.)

Pointers are an abstraction of machine addresses

A box-and-arrow diagram



represents at the hardware level

for some memory address `n`.

But in C, we are not supposed to care about actual hardware addresses.

The view in C of memory is a graph:

nodes = chunks of memory (often a struct)

edges = pointers

That is why box-and-arrow diagrams are so useful.

Why pointers are hard for everybody

- Pointers are hard because they are non-local
- imperative programming without pointers (like Basic/Fortran):
  $x = 2; y = y + 1;$
  The assignment to x does not change the value of y, and conversely.
- A graph that changes non-locally.
- The same issues arise in Java with references between objects.
- They may be obscured by Java bureaucracy.
- Pointers in C/C++ are made even harder by the need to manage memory.

Pointers and current research

- Pointers have been around for a long time, but have often been considered incomprehensible
- Since about 2000, there has been a huge amount of research on pointers, particularly Separation Logic.
  $\Rightarrow$ Program analysis tools in industry, e.g. Microsoft and Facebook (Infer).
- We will use Valgrind.

Pointers and pointer type in C: the * operator

- In C, `*` is also a unary operator.
- It has nothing to do with the binary infix operator for multiplication, even though it uses the same character.
- If `P` is an expression denoting a pointer, then `*P` is the result of dereferencing the pointer.
- If `T` is a type, then `T *p;` declares `p` to be of type "pointer to `T`"
- If `T` is a type, then `T*` is the type of pointers to something of type `T`. This is used for example in casting.
- pointer = programming abstraction of machine address in main memory
- dereferencing = load value from memory

The address operator & in C

- If a variable appears on the right-hand side of an =, its R-value is taken.
- If we want the address of `x` rather than its contents, we use the `&` operator, as in `&x`
- `y = x;` means `y` gets the contents of `x`
- `p = &x;` means `p` is made to point to `x`
- Quiz: what is `*&x`
- Note: in C++ `&` also used as the type constructor for references, e.g. `int&`.

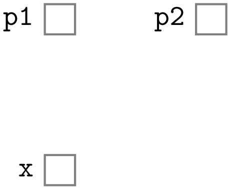
Pointer example: $*$, &, and aliasing

```
int x;
int *p1;
int *p2;
x = 5;
p1 = &x;
p2 = p1;
(*p2)++;
```


What is the value of `x` at the end?

Pointer example: ∗, &, and aliasing

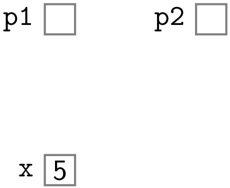
```
int x;
int *p1;
int *p2;
x = 5;
p1 = &x;
p2 = p1;
(*p2)++;
```



What is the value of `x` at the end?

Pointer example: $*$, &, and aliasing

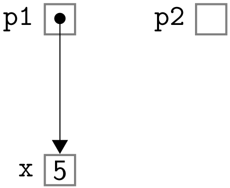
```
int x;
int *p1;
int *p2;
x = 5;
p1 = &x;
p2 = p1;
(*p2)++;
```



What is the value of `x` at the end?

Pointer example: ∗, &, and aliasing

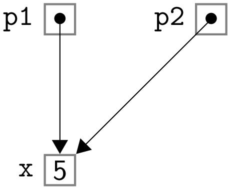
```
int x;
int *p1;
int *p2;
x = 5;
p1 = &x;
p2 = p1;
(*p2)++;
```



What is the value of `x` at the end?

Pointer example: $*$, &, and aliasing

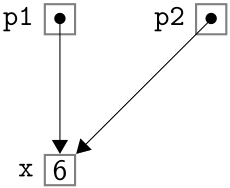
```
int x;
int *p1;
int *p2;
x = 5;
p1 = &x;
p2 = p1;
(*p2)++;
```



What is the value of `x` at the end?

Pointer example: $*$, &, and aliasing

```
int x;
int *p1;
int *p2;
x = 5;
p1 = &x;
p2 = p1;
(*p2)++;
```



What is the value of `x` at the end?

Pointer example - how not to think ☹

```
int x;
int *p1;
int *p2;
x = 5;
p1 = &x;
p2 = p1;
(*p2)++;
```

What about this reasoning:
$x = 5$
$p1 = 5$
$p2 = 5$
p2 updated, so
$p2 = 6$
Hence x is 5 at the end.

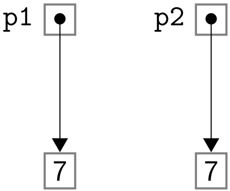
Pointer == in C

In C, two pointers are == if they refer to the same address in memory.
Pointer equality is different from structural equality like that built into functional languages, e.g., in OCaml:

```
# [ 1 ] = [ 1 ];;
- : bool = true
```

`p = q` makes `p == q`
`*p = *q` does not make `p == q`
In C++, you can overload ==.

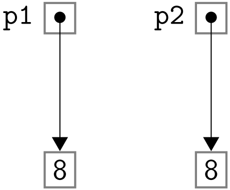
Pointer equality example 1



True or false?

`p1 == p2`

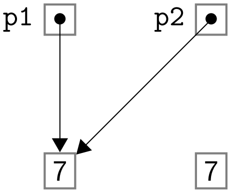
Pointer equality example 2



True or false?

```
*p1 == *p2
```

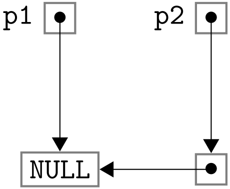
Pointer equality example 3



True or false?

```
p1 == p2
```

Pointer equality example 4



True or false?

`p1 == *p2`

`p2 == NULL`

`p1 == NULL`

`**p2 == NULL`

Exercise

Write Java code that shows the same issues as the previous pointer diagrams, in terms of aliasing, update, and equality.
In Java, you need to use object references instead of pointers.

Pointer type and declaration syntax

```
int *p, n;
```

is like

```
int *p;
int n;
```

and not

```
int *p;
int *n;
```

Pitfall: The `*` sticks only to the `p` and not the `int`.
That is why I don't write `int* p;` .
Array declarations behave the same, sticking only to one identifier.

Quiz

What are the types?

```
float **q, *p, a[], f;
```

What is wrong with this:

```
int *p, n;
p = n;
```

Exercise

Suppose

```
int *p1, *p2l
```

Explain the difference between

```
p1 = p2;
```

and

```
*p1 = *p2;
```

Exercise

```
int x, y, *p1, *p2, **q1, **q2;
x = 10;
y = 20;
p1 = &x;
q1 = &p2;
q2 = q1;
*q2 = p1;
(**q1) = 7;
```

What is the value of `x` at the end? Draw the memory with the pointers as arrows.

malloc and free from stdlib

- `stdlib.h` provides C functions `malloc` and `free`
- The part of memory managed by `malloc` is called the heap.
- `malloc`: you borrow some memory from the memory manager
- `free`: you give back the memory and promise not to touch it again
- The memory manager cannot force you to keep that promise; it is up to you
- if you use memory incorrectly in C, you get undefined behaviour
- malloc and free are not part of the C language itself, only its standard library
- You could implement your own malloc and free in C

malloc and free

The function `malloc` borrows some uninitialized memory in the heap from the memory allocator.

before: 


```
p = malloc(N);
```

after:

The function `free` gives memory back to the memory allocator.

before:


```
free(p);
```

after:

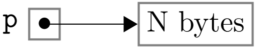
The call to `free` changes the ownership of the memory, not p or other pointers to it.

malloc and free

The function `malloc` borrows some uninitialized memory in the heap from the memory allocator.

before: 


```
p = malloc(N);
```

after: 

The function `free` gives memory back to the memory allocator.

before:


```
free(p);
```

after:

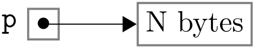
The call to `free` changes the ownership of the memory, not p or other pointers to it.

malloc and free

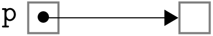
The function `malloc` borrows some uninitialized memory in the heap from the memory allocator.

before: 


```
p = malloc(N);
```

after: 

The function `free` gives memory back to the memory allocator.

before: 


```
free(p);
```

after:

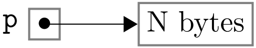
The call to `free` changes the ownership of the memory, not p or other pointers to it.

malloc and free

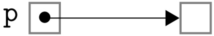
The function `malloc` borrows some uninitialized memory in the heap from the memory allocator.

before: 

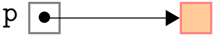

```
p = malloc(N);
```

after: 

The function `free` gives memory back to the memory allocator.

before: 

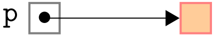

```
free(p);
```

after: 

The call to `free` changes the ownership of the memory, not p or other pointers to it.

Memory after freeing

Atfer `free(p)`, the memory is no longer owned by the program.



Dereferencing `p` causes undefined behaviour. The program may crash, or anything at all may happen, such as memory changing its content in unpredicatable ways.

malloc and free internally

A memory allocator could be implemented in C as follows:

- The allocator requests some memory from the OS (via sbrk in Unix)
- The available memory is divided into chunks that are linked together in a "free list"
- malloc detaches a chunk from the free list and returns a pointer to it
- free takes a pointer to a chunk and links it into the free list again
- problems: efficiency, memory fragmentation
- a naive allocator is in K&R
- Doug Lea's malloc is more sophisticated: `http://g.oswego.edu/dl/html/malloc.html`

How to think of deallocation

After free is called on some memory, various things may actually happen.

- The same piece of memory is re-used in a later malloc.
- The memory manager writes its own data structures into the memory (e.g. free list).

Rather than trying to guess what exactly happens, we call all of this undefined behaviour.
C (unlike Java) does not prevent you from doing bad things.
You can still access the memory but should not.
One could think of the memory as “cursed”, so to speak.

sizeof operator

- For using `malloc`, we need to the function how many bytes to allocate.
- Usually enough to hold value of some type.
- But sizes are implementation dependent.
- The compiler tells us how big it makes each type.
- `sizeof(T)` gives the size in bytes for some type `T`.
- Hence the idiom

```
T *p = malloc(sizeof(T))
```

for some type `T`.

calloc and realloc

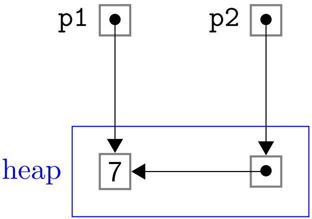
stdlib.h also contains these variants of `malloc`:

`calloc` allocate and initialize to zeros

`realloc` reallocate

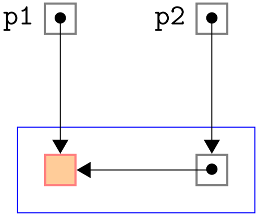
malloc example

```
int *p1, **p2;
p1 = malloc(sizeof(int));
*p1 = 7;
p2 = malloc(sizeof(int*));
*p2 = p1;
```



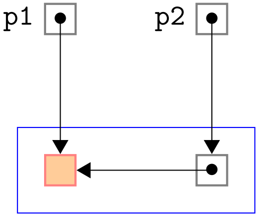
malloc and free example

```
int *p1, **p2;
p1 = malloc(sizeof(int));
*p1 = 7;
p2 = malloc(sizeof(int*));
*p2 = p1;
free(p1);
```



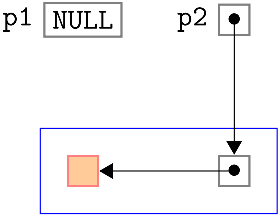
use after free example ☹

```
int *p1, **p2;
p1 = malloc(sizeof(int));
*p1 = 7;
p2 = malloc(sizeof(int*));
*p2 = p1;
free(p1);

**p2 = 11;
```



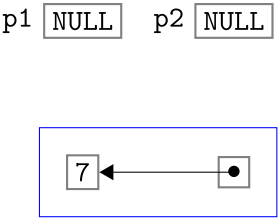
double free example ☹

```
int *p1, **p2;
p1 = malloc(sizeof(int));
*p1 = 7;
p2 = malloc(sizeof(int*));
*p2 = p1;
free(p1);
p1 = NULL;
free(*p2);
```



Memory leak example ☹

```
int *p1, **p2;
p1 = malloc(sizeof(int));
*p1 = 7;
p2 = malloc(sizeof(int*));
*p2 = p1;
p1 = NULL;
p2 = NULL;
```



Exercise

Draw the memory after the following code has run.

```
int *p1, *p2, **q;
p1 = malloc(sizeof(int));
p2 = malloc(sizeof(int));
q = malloc(sizeof(int*));
*q = p1;
**q = 7;
*q = p2;
free(*q);
free(q);
```

Structures in C

- A structure (or struct) in C is much like a Java class that contains only data (no methods)
- C++ jargon: POD for plain old data
- evolution: C struct $\rightarrow$ C++ class $\rightarrow$ Java class
- A structure contains members. (Not variables)
- In C++ (not plain C), you can also define functions inside a struct
- This gives OO, where operations on data are packaged together with the data in objects
- In C, functions are defined outside structs and often access them via pointers
- Structures and pointers (along with malloc) let us build many classic data structures: lists, trees, graphs

Structure syntax

C structure syntax is similar to Java class syntax:

```
struct s {
    T1 m1;
    ...
    Tk mk;
};
```

Here `T1, ..., Tn` are type names.
After a structure s has been declared, `struct s` can be used as a type name.

```
int n;         // declares n as an int
struct s y;    // declares y as a struct s
struct s *p;   // declares p as a pointer to a struct s
```

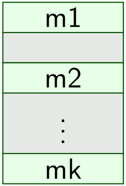
Access to structure member is by using the dot operator, e.g., `s.m1`.

Structure syntax example

```
struct Point {
    int x, y;
};
...
struct Point v;
v.x = v.y;
```

Structure layout in memory

```
struct S {
    T1 m1;
    T2 m2;
    ...
    Tk mk;
};
```

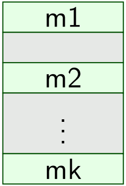


Structure members are laid out in memory in order.
There may be a few bytes of padding between structure members due to alignment, depending on the hardware.
This may get some padding to get the pointer aligned:

```
struct S {
    char c;
    char *p;
};
```

Structure and sizeof

```
struct S {
    T1 m1;
    T2 m2;
    ...
    Tk mk;
};
```

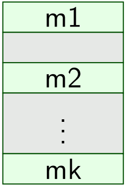


What is `sizeof(struct S)`?

$\texttt{sizeof(struct S)} \geq \texttt{sizeof(T1)} + \ldots + \texttt{sizeof(Tk)}$

Structure and sizeof

```
struct S {
    T1 m1;
    T2 m2;
    ...
    Tk mk;
};
```



What is `sizeof(struct S)`?

$$\texttt{sizeof(struct S)} \geq \texttt{sizeof(T1)} + \ldots + \texttt{sizeof(Tk)}$$

Recursive types from structures and pointers

Standard example of recursive data structures: list and trees.

```
struct IntList {
    struct IntList *next;
    int data;
};

struct Bintree {
    struct BinTree *left, *right;
    int data;
};

struct Quadtree {
    struct QuadTree *chld[4];
    int data;
};
```

The pointers are required for the recursion.

`->` operator

- `p->m` is an abbreviation for `(*p).m`.
- Dereference pointer, then structure member access.
- Very common in C and C++ code.
- Useful for chaining together:

```
p->m1->m2->m3
```

- Also used for OO (member functions) in C++, as in `p->f()`
- Exercise: write `p->x->y->z` using only `.` and `*`.

List traversal idiom in C

A pointer `p` in a condition

```
while(p) { ... }
```

is equivalent to

```
while(p != NULL) { ... }
```

It is a common idiom for looping over lists, along with an assignment such as

```
p = p->next;
```

In modern C++ and Java, one could use iterators for this situation, but since C does not have an iterator construct, one uses the idiom above.

Example: deleting all elements of a linked list

```
  while(lp) {
    q = lp;
    lp = lp->next;
    free(q);
}
```

Why do we need the extra pointer q? Why not

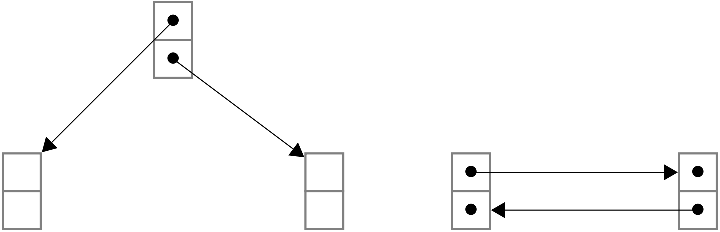
```
  while(lp) {
    free(lp);
    lp = lp->next;
}
```

Traversal and recursion vs while

- For lists, a `while` loop is sufficient to traverse
- A modern C compiler may, but is not required to, compile tail recursion as efficiently as a while loop.
- For traversing trees and graphs, recursion is much easier to program correctly than using a while loop.
- It is always possible to write the same code without recursion, but possibly using extra data structures

Same stucture used for binary trees and doubly-linked lists

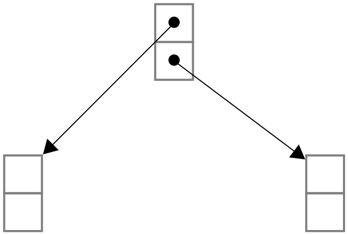
```
struct twoptrs {
  struct twoptrs *one, *two;
};
```



Traversing or deleting is very different for trees or doubly linked lists.

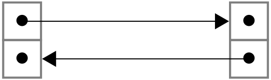
Tree

```
struct twoptrs {
  struct twoptrs *one, *two;
};
...
struct twoptrs *p1 = malloc(sizeof(struct twoptrs));
struct twoptrs *p2 = malloc(sizeof(struct twoptrs));
struct twoptrs *p3 = malloc(sizeof(struct twoptrs));
p3->one = p1;
p3->two = p2;
```



Doubly-linked list

```
struct twoptrs {
  struct twoptrs *one, *two;
};
...
struct twoptrs *p1 = malloc(sizeof(struct twoptrs));
struct twoptrs *p2 = malloc(sizeof(struct twoptrs));
p1->one = p2;
p2->two = p1;
```



Exercise

Draw the memory produced by the following code.

```
struct twoptrs {
  struct twoptrs *ptrone, *ptrtwo;
}

struct twoptrs *p1, *p2, *p3;

p1 = malloc(sizeof(struct twoptr));
p2 = malloc(sizeof(struct twoptr));
p3 = malloc(sizeof(struct twoptr));

p1->ptrone = NULL;
p1->ptrtwo = NULL;
p2->ptrone = NULL;
p2->ptrtwo = NULL;
p3->ptrone = p1;
p3->ptrtwo = p2;
p1->ptrone = p3;
```

Exercise

Draw the memory produced by the following code.

```
struct twoptrs {
  struct twoptrs *ptrone, *ptrtwo;
}

struct twoptrs *p1, *p2, *p3;

p1 = malloc(sizeof(struct twoptr));
p2 = malloc(sizeof(struct twoptr));
p3 = malloc(sizeof(struct twoptr));

p1->ptrone = NULL;
p1->ptrtwo = p2;
p2->ptrone = p1;
p2->ptrtwo = p3;
p3->ptrone = p2;
p3->ptrtwo = NULL;
```

Example: doubly linked list traversal

```
struct doublylinked {
    int data;
    struct doublylinked *next;
    struct doublylinked *prev;
};

void printdl(struct doublylinked *p)
{
    while(p) {
        printf("%10d", p->data);
        p = p->next;
    }
    printf("\n");
}
```

Exercise: write the above using recursion rather than `while`.

Example: doubly linked list item deletion

```
struct doublylinked {
    int data;
    struct doublylinked *next;
    struct doublylinked *prev;
};

void removedl(struct doublylinked *p)
{
    if(!p) return;
    if(p->prev)
        p->prev->next = p->next;
    if(p->next)
        p->next->prev = p->prev;
    free(p);
}
```

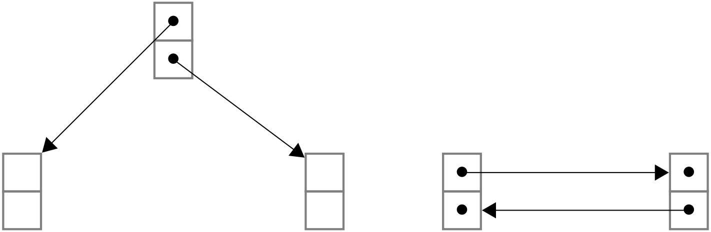
Example: some complicated structs in Doug Lea's malloc

```
struct malloc_chunk {
  size_t              prev_foot;  /* Size of previous chunk (if
  size_t              head;       /* Size and inuse bits. */
  struct malloc_chunk* fd;        /* double links -- used only
  struct malloc_chunk* bk;
};

struct malloc_tree_chunk {
  /* The first four fields must be compatible with malloc_chunk
  size_t                    prev_foot;
  size_t                    head;
  struct malloc_tree_chunk* fd;
  struct malloc_tree_chunk* bk;

  struct malloc_tree_chunk* child[2];
  struct malloc_tree_chunk* parent;
  bindex_t                  index;
};
```

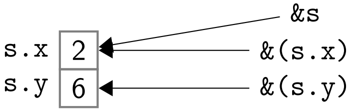
From ftp://g.oswego.edu/pub/misc/malloc.c

Each node is part of both a tree and a doubly-linked list

```
struct malloc_tree_chunk {
  struct malloc_tree_chunk* fd;
  struct malloc_tree_chunk* bk;

  struct malloc_tree_chunk* child[2];
  struct malloc_tree_chunk* parent;
};
```



Addresses of structure members



Structures inside structures

What is the difference between `B1` and `B2`?

Draw the memory layout.

Compare `sizeof(struct B1)` and `sizeof(struct B2)`.

```
struct A {
    long x[80];
};

struct B1 {
    struct A a;
};

struct B2 {
    struct A *p;
};
```

Quiz

Consider

```
struct s {
  int x;
  int y;
};
```

```
struct s a;
```

True of false?

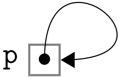
```
&a == &(a.x)
```

True of false?

```
&a == &(a.y)
```

Exercise

Is it possible for a pointer to point to itself, like this:

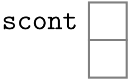


If yes, write the code.

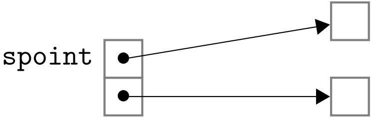
There may be more or less clean ways to do it.

Structures containing structures or pointers to them

```
struct scont {
  A a;
  B b;
};
```




```
struct spoint {
  A *ap;
  B *bp;
};
```



typedef

```
typedef
struct s {
...
} t1;

typedef struct s t2;

t1 *p;
```

We won't use typedef in the module. (Torvalds does not like it either, see Linux code.)
In C++, there is better syntax.

In C, no function inside structs

```
struct S {
   int x, y;
};

void setx(struct C *p, int n)
{
  p->x = n;
}
```

Object oriented in C++, like Java

```
struct S {
   int x, y;
   void setx(int n) { x = n; }
};
```

Extending structs

```
struct SinglyLinked {
  struct SinglyLinked next;
  int data;
};

struct DoublyLinked {
  struct DoublyLinked *n;
  int data;
  struct DoublyLinked *p;
};
```

what about:

```
struct DoublyLinked {
  int data;
  struct DoublyLinked *next;
  struct DoublyLinked *prev;
};
```

Valgrind and memcheck

`http://valgrind.org`
`http://valgrind.org/docs/manual/quick-start.html`

> *Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail.*

`http://valgrind.org/docs/manual/mc-manual.html`
In C, the memoy is not actually red.
O RLY? YA RLY.
Rather, it becomes nondeterministic.
Valgrind makes the red memory observable and produces errors.
Likewise for memory leaks.

Using clang and valgrind

For background on Memcheck, see
`http://valgrind.org/docs/manual/mc-manual.html`.
On the Linux lab machines:

```
module load llvm
```

Suppose your program is called `frodo.c`.

```
clang -o frodo frodo.c
valgrind --leak-check=full ./frodo
```

Strings in C

- In C a string is an array of char terminated by a zero byte
- Zero byte `\0` is not the same as the character for “0” (which is 48 in ASCII).
- The size of the array is not stored (unlike Java).
- You need to keep track of array bounds yourself
- When an array is passed to a function, a pointer to the start of the array is passed, not the contents of the array

Pointer arithmetic and arrays

- In C, you can add a pointer and an integer
- You cannot add two pointers
- Array access is via pointer arithmetic
- Pointer arithmetic is typed
- `p + 1` does not mean `p` plus one byte
- in `p + n`, `n` is scaled up by the size of the type of what `p` points to
- array indexing

  ```
  a[i]
  ```

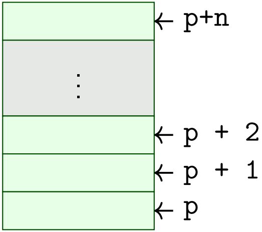
  is shorthand for

  ```
  *(a + i)
  ```

- Implemented via indexed addressing

Pointer arithmetic

↑ higher addresses



Each of the cells is `sizeof(T)` wide if `p` is of type `T`

Pointer arithmetic is automagically scaled by the type system

Prefix and postfix operator precedence

In C, postfix operators bind more tightly than prefix ones.

```
*p++
```

is parsed like

```
*(p++)
```

Similarly

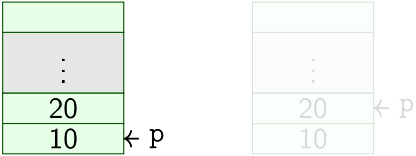
```
*f()
```

is parsed like

```
*(f())
```

p++ vs (*p)++

p++ increments the pointer p:

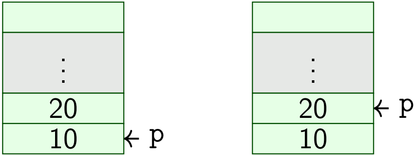


*p++ gives the value of *p before incrementing p, in this case 10.

(*p)++ increments the value pointed to by p:

p++ vs (*p)++

p++ increments the pointer p:

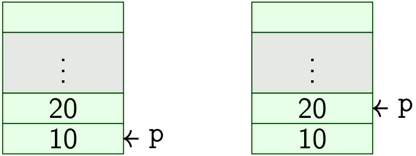


*p++ gives the value of *p before incrementing p, in this case 10.

(*p)++ increments the value pointed to by p:

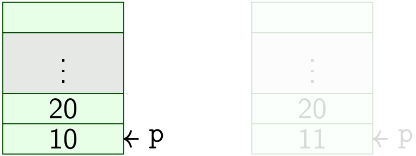
p++ vs (*p)++

p++ increments the pointer p:



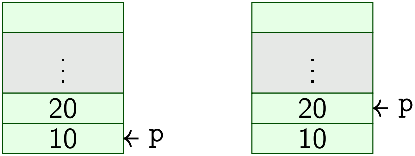
*p++ gives the value of *p before incrementing p, in this case 10.

(*p)++ increments the value pointed to by p:



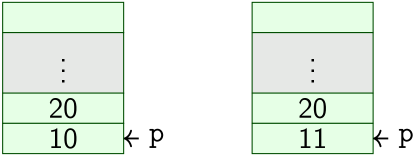
p++ vs (*p)++

p++ increments the pointer p:



*p++ gives the value of *p before incrementing p, in this case 10.

(*p)++ increments the value pointed to by p:



Exercise

What does this do:

```
int a[10], *p;

p = a + 2;

p++;

(*p)--;

--*p;

*--p;

*p = *p * *p;
```

String copy idiom from Kernighan and Ritchie 😎

```
while(*p++ = *q++);
```

This is typical C code, `*p++` etc

Kernighan and Ritchie: an idiom that should be mastered

Unbounded copy is the cause for many, very severe security vulnerabilities: buffer overflow

String copy example ☺

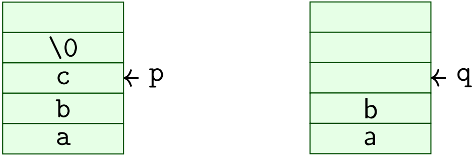
```
void mystrcpy(char *q, char *p)
{
    while(*q++ = *p++);
}
...
char from[] = "abc";
char to[4];
mystrcpy(to, from);
```

String copy example ☺

```
void mystrcpy(char *q, char *p)
{
    while(*q++ = *p++);
}
...
char from[] = "abc";
char to[4];
mystrcpy(to, from);
```

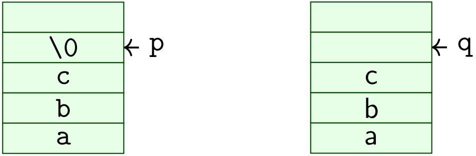
String copy example ☺

```
void mystrcpy(char *q, char *p)
{
    while(*q++ = *p++);
}
...
char from[] = "abc";
char to[4];
mystrcpy(to, from);
```



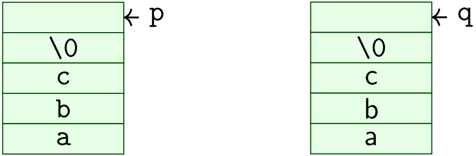
String copy example ☺

```
void mystrcpy(char *q, char *p)
{
    while(*q++ = *p++);
}
...
char from[] = "abc";
char to[4];
mystrcpy(to, from);
```



String copy example ☺

```
void mystrcpy(char *q, char *p)
{
    while(*q++ = *p++);
}
...
char from[] = "abc";
char to[4];
mystrcpy(to, from);
```



String copy overflow ☹

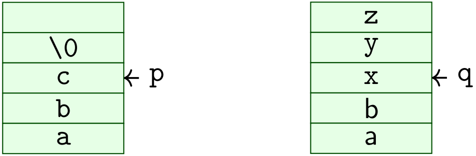
```
void mystrcpy(char *q, char *p)
{
    while(*q++ = *p++);
}
...
char a[] = "abc";
char b[2];
mystrcpy(b, a);
```

String copy overflow ☹

```
void mystrcpy(char *q, char *p)
{
    while(*q++ = *p++);
}
...
char a[] = "abc";
char b[2];
mystrcpy(b, a);
```

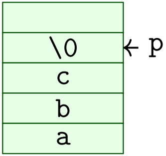
String copy overflow ☹

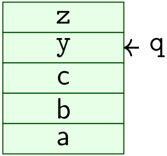
```
void mystrcpy(char *q, char *p)
{
    while(*q++ = *p++);
}
...
char a[] = "abc";
char b[2];
mystrcpy(b, a);
```



String copy overflow ☹

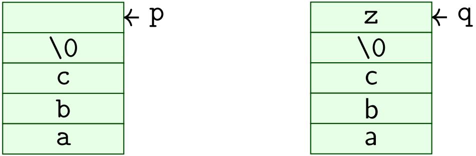
```
void mystrcpy(char *q, char *p)
{
    while(*q++ = *p++);
}
...
char a[] = "abc";
char b[2];
mystrcpy(b, a);
```





String copy overflow ☹

```
void mystrcpy(char *q, char *p)
{
    while(*q++ = *p++);
}
...
char a[] = "abc";
char b[2];
mystrcpy(b, a);
```



Buffer overflow on the call stack ☹

```
int vulnerable_function()
{
    int winner = 0; // suppose this is security-critical
    char name[8];   // this is the buffer to be overflown

    printf("Please enter your name:\n");
    fgets(name, 200, stdin); // too much input
    ...
}
```

Input `blahblahbl` overflows the string variable on the stack:

	return address
`bl\0`	winner
`blahblah`	name

Note: the call stack grows towards lower machine addresses.

Buffer overflow prevention and mitigation

- All array accesses in C are potentially dangerous
- Strings in C are arrays and can overflow
- “All input is evil”
- Check bounds for arrays
- Use functions with bounds such as `strncpy`, `fgets`
- Watch out for off-by-one errors in bounds
- C compilers do some buffer overflow mitigation (stack canaries)
- For more, see Seacord, “Secure Programming in C and C++”
- For advanced attacks, you need to understand how C is compiled (call stack, return address, etc)

Array access in C vs Java

Java does automatic bounds check, C does not.
In Java, `a[i]` either

1. refers to the i-th element of array a ☺
2. throws an out-of-bounds exception if i is too big ☹

In C, `a[i]` either

1. refers to the i-th element of array a ☺
2. causes undefined behaviour if i is too big ☹

Example:

```
int a[5];
a[9999999999] = 666;
// segfault likely, but anything might happen
```

Arguments to main = array and count

`main` takes two arguments: an array of strings and the number of strings. The command line arguments are passed this way.

```
main(int argc, char *argv[]) { ... }
```

```
main(int argc, char **argv) { ... }
```

If we do not need the arguments, we can also write in C (but not C++)

```
main() { ... }
```

Exercise: write a main function that prints out its command line arguments using

```
printf("%s\n", argv[i]);
```

Stack data structure = array + stack pointer

```
push(10);
push(20);
push(30);
```

Stack drawn as growing upward:

	← `stack + stacksize - 1`
⋮	
	← `stackptr`
30	← `stack` + 2
20	← `stack` + 1
10	← `stack` = `&stack[0]`

Push and pop using pointer arithmetic

```
int stack[100];

int *sp = stack;

void push(int n)
{
    *sp++ = n;
}

int pop()
{
    return *--sp;
}
```

Push onto stack with bounds check

```
int stack[100];

int *sp = stack;

// invariant: sp points to first free element

void push(int n)
{
    if (sp < stack + stacksize - 1)
        *sp++ = n;
    else {
        fprintf(stderr, "Stack overflow!\n\n");
        exit(1);
    }
}
```

Pop from stack with bounds check

```
int stack[100];

int *sp = stack;

// invariant: sp points to first free element

int pop()
{
    if (sp > stack)
        return *--sp;
    else {
        fprintf(stderr, "Stack underflow!\n\n");
        exit(1);
    }
}
```

Example: using a stack to evaluate arithmetic expressions

```
while(1) {
    fgets(input, inputbufsize, stdin);
    switch(input[0])
    {
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            push(atoi(input));
            break;
        case '+':
            push(pop() + pop());
            break;
        case '-':
            push(pop() - pop());
            break;
        default:
            printf("Goodbye.\n\n");
            return 0;
    }
```

http://www.cs.bham.ac.uk/~hxt/2015/c-plus-plus/StackEval.c

Exercise

Rewrite the stack operations without pointer arithmetic, using array indexing instead.

Void pointers as a universal pointer type

- void pointers are a feature of the C type system
- void pointers are a bit of a hack. C++ templates are much cleaner and more powerful, but not available in C.
- a void pointer should never be dereferenced
- a void pointer can be cast to and from any pointer type
- this has nothing to so with what the pointer points to at run time.
- in C (but not C++), casts from void pointer may be left implicit.
- `malloc` returns a void pointer
- `free` takes a void pointer as an argument
- nice C code contains few casts except the implicit ones in `malloc` and `free` ☺

Void pointer example

```
void *vp;  // vp is declared as void pointer
int *ip;   // ip is declared as an int pointer
vp = malloc(sizeof(int));  // vp now points into memory
ip = vp;             // implicit cast from void pointer
ip = (int*)vp;       // explicit cast from void pointer
free(ip);            // this does NOT make ip a void pointer
ip = NULL;           // neither does this
*vp = 42;            // type error, void not int
```

Pointers and casting

Pointer casting does not change what is pointed at.
Compare and contrast: cast from int to float.

```
int x;
float f;
x = 10;
f = (float)x; // f is 10.0
```

```
int x;
float *fp;
x = 10;
fp = (float*)&x; // *fp is nonsense, not (float)x
```

Pointer casting does not perform any dynamic checks.
Compare and contrast: casting between objects in Java.

Quiz

```
T *p;

p + 1 == (T*)((char*)p + 1)
```

True or not?

Quiz

What is wrong with this:

```
int x;
// some more code
free(&x);
```

Is there a type error?

First-class functions pointer in C and C++

- C has pointers to functions.
- In C, functions *cannot* be defined inside other functions.
- Functions in C can be passed as parameter very easily: they are just code pointers.
- Note: C++11 has lambda expressions and a general function type.

Parsing `int *f(int)`

Inside out from identifier:

`f`

`f(int)`

`*f(int)`

`int *f(int)`

Function with an `int` parameter and returning a pointer to an `int`

Parsing `int (*f)(int)`

Inside out from identifier, not left to right

`f`

`*f`

`(*f)(int)`

`int (*f)(int)`

Pointer to function taking an `int` parameter and returning an `int`

Exercise

What are these types in English?

```
struct s *f(int)
```

```
struct s *(*f)(struct s (*)(int))
```

```
int (*)(int, int)
```

Example of function pointer: fold function

A binary operator is passed as a function pointer argument.

$$\texttt{fold}\ n \oplus [x_1, \ldots x_n] = n \oplus x_1 \oplus \cdots \oplus x_n$$

```
int fold(int n, int (*bin)(int, int),
        struct Linked *p)
{
    while (p) {
        n = bin(n, p->data);
        p = p->next;
    }
    return n;
}
```

Example of function pointer: sort function

Quicksort from C library. A comparison function is passed as a function pointer argument.

```
void qsort (void* base, size_t num, size_t size,
            int (*compar)(void*, void*));
```

Comparison function using void pointers:

```
int comparefloat (void *p, void *q)
{
  if ( *(float*)p <  *(float*)q ) return -1;
  if ( *(float*)p == *(float*)q ) return 0;
  if ( *(float*)p >  *(float*)q ) return 1;
}
```

Example of function pointer: recursion via pointer 😎

```
int (*fp)(int); // function pointer as global variable

int facnonrec(int n)
{
    if(n == 0)
        return 1;
    else
        return n * (*fp)(n - 1); // no recursion
}

int main(int argc, char *argv[]) {
    fp = facnonrec;  // make recursion via pointer
    printf("%d\n", facnonrec(5)); // 120
}
```

Exercise

Exercise: rewrite the fold function with void pointers so that it works for arbitrary types (like the qsort function) and not only integers.

C pointers vs Java references

Object references in Java are similar to C pointers, but safer.
A Java reference either

1. is equal to null, or
2. refers to something we can access in memory

A C pointer

1. is equal to NULL, or
2. points to something we can access in memory, or
3. points to something we should not access, as it may cause undefined behaviour
   example: `p` after `free(p);`
   example: `a[i] = 2;` if `n` is out of bounds

The third possibility makes a huge difference between memory-safe languages like Java (and OCAML) and and unsafe languages like C and C++.

C nondeterminism vs Java determinism

So when you have a bug

- Java gives you exceptions
- C/C++ gives you segmentation faults

What's the big deal?

C may give you a segfault. It does not have to.

Undefined behaviour includes silently changing values in anywhere in memory.

May be different every time you run the code.

Have fun debugging ...

Valgrind to the rescue!

Conclusion of pointers in C part of the module

You have seen the part of C most relevant to systems programming:

1. pointer types and operations ✓
2. pointer equality ✓
3. malloc and free ✓
4. structures and pointers ✓
5. pointer arithmetic ✓
6. strings and pointers ✓
7. function pointers ✓

Syntax:

```
* & = == malloc free struct . -> ++ -- (*f)()
```

Conclusions

Once you understand pointers in C, they make sense.