Numbers in C

Balance Sheet … so far

Lost	***Gained***
• Contracts • Safety • Garbage collection • Memory initialization • Well-behaved arrays • Fully-defined language • Strings	• Preprocessor • Undefined behavior • Explicit memory management • Separate compilation • Pointer arithmetic • Stack-allocated arrays and structs • Generalized address-of

Undefined Behavior

Memory	• Reading/writing to non-allocated memory • Reading uninitialized memory • even if correctly allocated • Use after free • Double free • Freeing memory not returned by malloc/calloc • Writing to read-only memory
Numbers	**Today**

The type int

int Sizes

- In C0/C1, the size of values of type int is 32 bits
  - and pointers are 64 bits
- In C, the size of an int has evolved over time
  - and pointers too

	'70s	80's	'90s	Today
Pointer size	8	16	32	64
int size	8	16	32	32

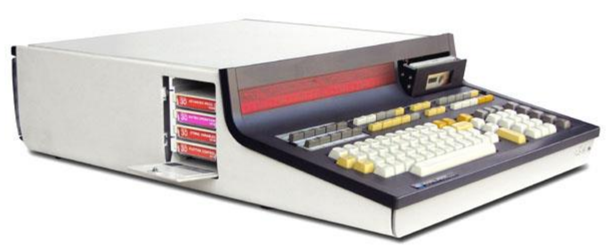
Typical

int Sizes

- In C, the size of an int has evolved over time
  - and pointers too

Pointer size	**8**	16	32	64
int size	**8**	16	32	32
	‘70s	80’s	‘90s	Today

HP 9830A



- Early computers had **8-bit** addresses
  - 256 ***bytes*** of memory
    - RAM was very expensive
  - ints ranged from -128 to 127

‘60s

The computer that sent Apollo 11 to the moon

int Sizes

- In C, the size of an int has evolved over time
  - and pointers too

Pointer size	8	**16**	32	64
int size	8	**16**	32	32
	‘70s	**80’s**	‘90s	Today

Commodore 64

- 16-bit addresses
  - (up to) 64 kilobytes of memory
    - the Commodore 64
  - ints ranged from -32768 to 32767

Apple II

int Sizes

- In C, the size of an int has evolved over time
  - and pointers too

Pointer size	8	16	**32**	64
int size	8	16	**32**	32
	‘70s	80’s	**‘90s**	Today

iMac

PC

- 32-bit addresses
  - (up to) 4 gigabytes of memory
  - ints ranged in the billions

int Sizes

- In C, the size of an int has evolved over time
  - and pointers too

	'70s	80's	'90s	**Today**
Pointer size	8	16	32	**64**
int size	8	16	32	**32**

- 64-bit addresses
  - nobody has 2^{64} bytes memory
  - billions are still Ok for ints

Implementation-defined Behavior

- The C standard says that it is for the compiler to define the size of an int
  - with some constraints

- It is **implementation-defined**

  The compiler decides, but
  - it remains fixed
  - the programmer can find out how big an int is
    - the file <limits.h> defines the values of INT_MIN and INT_MAX
      - and therefore the size of an int

Undefined behavior ≠ implementation-defined behavior

- undefined behavior does not have to be consistent
- the programmer has no way to find out from inside the program

Implementation-defined Behavior

- Most programmers don’t need to know how big an int is
  - just write code normally, possibly using INT_MIN and INT_MAX
  - the compiler will use whatever internal size it has chosen

This is not true of code that uses the bits of an int to encode data: bit patterns (e.g., pixels)

- Same thing for pointers

- Code written in the 1970s still works on today’s computers
  - as long as the code doesn’t depend on the size of an int
  - and the programmer used sizeof inside malloc

int's Undefined Behaviors

- **Safety violations in C0 are undefined behavior in C**
  - division/modulus by 0, or INT_MIN divided/mod'ed by -1
  - shifting by more than the size of an int

- **Overflow!**
  - C programs do not necessarily use two's complement
    - this makes it essentially impossible to reason about ints in a C program
    - an optimizing compiler cannot simplify $n + n - n$ to n
  - gcc provides the flag **-fwrapv** to force the use of two's complement for ints

Other Integer Types

Signed Integer Types

- C0 has a single type of integers: int

- C has many more
  - long: integers that are larger than int
    - 64 bits nowadays
  - short: integers that are smaller than int
    - 16 bits nowadays
  - char: integers that are smaller than short
    - 8 bits nowadays
    - but always 1 byte

char is a number!
- 'a' is convenience syntax
- the placeholder %c in printf displays it as a character

C99 defines a byte as *at least* 8 bit

  - … and there are more

Unsigned Integer Types

- Lots of code doesn't use negative numbers
- C provides **unsigned** variants of each integer type
  - same number of bits but sign bit can be used to represent more numbers
    - twice as many numbers
  - unsigned long
  - unsigned int
  - unsigned short
  - unsigned char

or just unsigned

- Overflow on unsigned numbers **is** defined to wrap around
  - unsigned numbers do follow the laws of modular arithmetic

Unsigned Integer Types

- size_t is the size of a pointer
  - the argument of malloc and calloc
  - array indices
  - return type of sizeof
  - …

Implementation-defined Integers

Whether char is signed or unsigned
is implementation-defined

signed	unsigned	C99 constraints	*Today's size*
signed char	unsigned char	exactly 1 byte	*8 bits*
short	unsigned short	range at least $(-2^{15}, 2^{15})$	*16 bits*
int	unsigned int	range at least $(-2^{15}, 2^{15})$	*32 bits*
long	unsigned long	range at least $(-2^{31}, 2^{31})$	*64 bits*
	size_t		*64 bits*

and there are several more …

Casting Integers

Integer Casts

- We go back and forth between different number types with **casts**

```
int x = 3;
long y = (long)x;
```

x is 0x00000003

y is 0x0000000000000003

- Literal numbers have always type int

```
3
```

this is an int

  - The compiler introduces **implicit casts** as needed

```
long x = 3;
```

    - is implicitly turned into

```
long x = (long)3;
```

Integer Casts

- Literal numbers have always type int
- The compiler introduces **implicit casts** as needed

- This can lead to unexpected outcome

  long x = 1 << 40;

  is *undefined behavior*

  - This is implicitly turned into

    long x = (long)(1 << 40);

1 is an int

This shift 1 by 40 positions

but 1 has only 32 bits

- Fix: long x = ((long)1) << 40;

Casting Rules

- When casting between signed and unsigned integers *of the same size*, the **bit pattern is preserved**

This is actually implementation-defined (but commonplace)

  - Example 1

```
signed char x = 3;                       // x is 3 (= 0x03)
unsigned char y = (unsigned char)x;      // y is 3 (= 0x03)
```

  - Example 2

```
signed char x = -3;                      // x is -3   (= 0xFD)
unsigned char y = (unsigned char)x;      // y is 253 (= 0xFD)
```

Casting Rules

- When casting small to big integers *of the same signedness*, the **value is preserved**

  - Example 1

```
signed char x = 3;          // x is 3 (= 0x03)
int y = (int)x;             // y is 3 (= 0x00000003)
```

It does **sign extension**

  - Example 2

```
signed char x = -3;         // x is -3 (= 0xFD)
int y = (int)x;             // y is -3 (= 0xFFFFFFFD)
```

Casting Rules

- When casting big to small integers *of the same signedness*, the **value is preserved** *if it fits*
  - the behavior is undefined otherwise

  - Example 1

```
int x = 3;                              // x is 3 (= 0x00000003)
signed char y = (signed char)x;         // y is 3 (= 0x03)
```

  - Example 2

```
int x = -3;                             // x is -3 (= 0xFFFFFFFD)
signed char y = (signed char)x;         // y is -3 (= 0xFD)
```

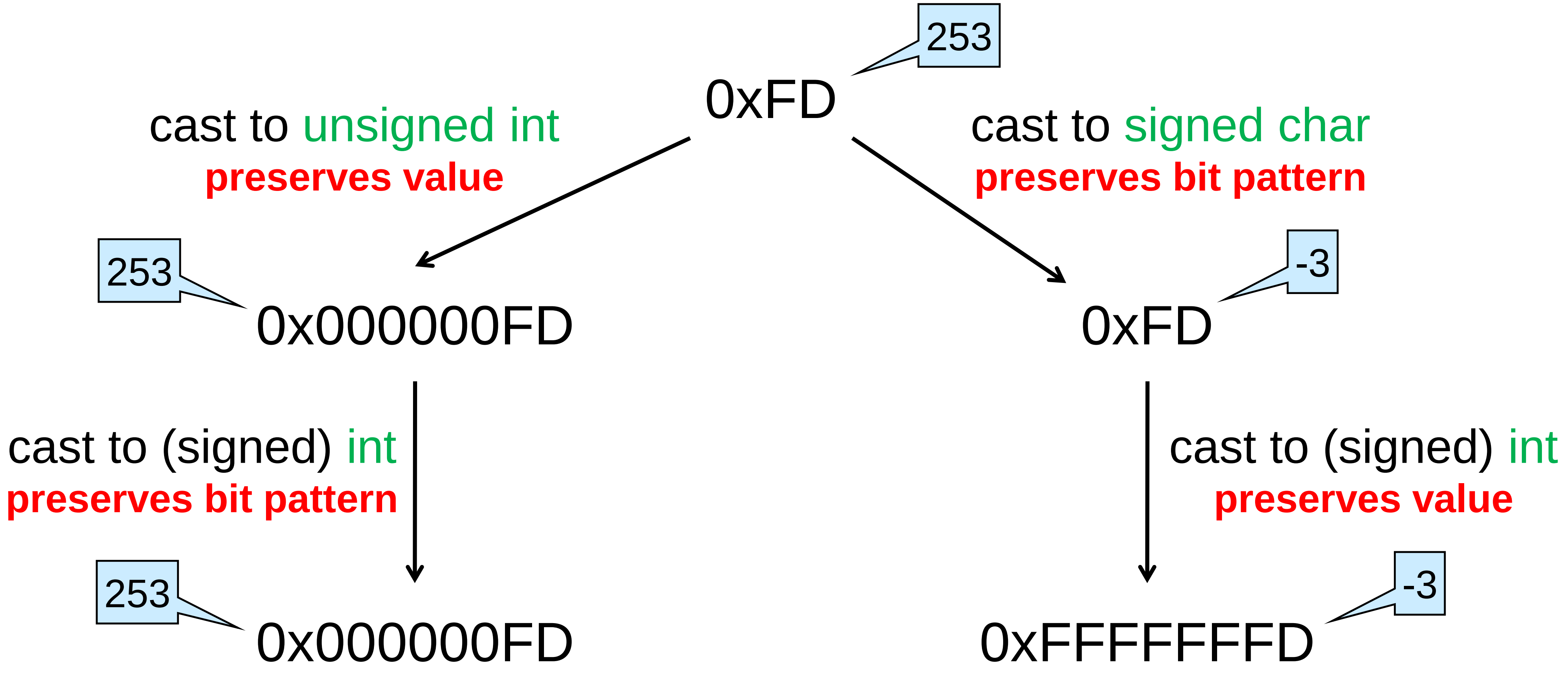
  - Example 3

```
int x = INT_MAX;                        // x is -3 (= 0x7FFFFFFF)
signed char y = (signed char)x;         // y is ??
```

Casting across Signedness and Size

- The compiler may apply the rules in either order

```
unsigned char x = 0xFD;          // x is 253
int y = (int)x;                  // y is …
```



- *is y 253 or -3?*

Casting across Signedness and Size

- The compiler may apply the rules in either order

```
unsigned char x = 0xFD;        // x is 253
int y = (int)x;                // y is …
```

  - Is y -3 or 253?
    - the order of casts is actually defined
      - but who remembers it?

- Solution: **be explicit**
  - Write either

```
int y = (int)(unsigned int)x;  // y is 253
```

    to change first the size and then the signedness
  - or

```
int y = (int)(signed char)x;   // y is -3
```

    to change first the signedness and then the size

Fixed-size Numbers

Fixed-size Integers

- For bit patterns, the program needs the number of bits to remain the same as C evolves

- Header file <stdint.h> provides **fixed-size integer types**
  - in signed and unsigned variants

Fixed-size signed	**Today's signed equivalent**	**Today's unsigned equivalent**	**Fixed-size unsigned**
int8_t	signed char	unsigned char	uint8_t
int16_t	short	unsigned short	uint16_t
int32_t	int	unsigned int	uint32_t
int64_t	long	unsigned long	uint64_t

That's the number of bits

Floating Point Numbers

float

- The type float represents **floating point numbers**
  - ➢ nowadays 32 bits

float x = 0.1;

float y = 2.0235E-27;

Numbers with a decimal point

That's 2.0235 * 10^{-27}

- float and int use the same number of bits, but float has a much larger range
  - ○ some numbers with a decimal point are not representable
  - ○ the larger range comes at the cost of **precision**
    - ➢ operations on floats may cause **rounding errors**

float

Danger

- Operations on floats may cause **rounding errors**
  - Example 1

```
#include <math.h>
#define PI 3.14159265
float x = sin(PI);
```

Defines sin, cos, log, …

Any more decimals would be ignored

In math, *sin(π)* is *0* but sin(PI) is not 0.0

  - Example 2

```
float y = (10E20 / 10E10) * 10E10;
```

That’s $(10^{20}/10^{10}) * 10^{10}$

- ➢ we expect y to be equal to 10E20
- ➢ but it isn’t always
  - ❑ it depends on the compiler

float

- Operations on floats may cause **rounding errors**
  - Example 3

```
for (float res = 0.0; res != 5.0; res += 0.1)
    printf("res = %f\n", res);
```

  - we expect the loop to terminate after 50 iterations
  - instead it runs for ever
  - That's because 0.1 decimal is a **periodic** number in binary: $0.0\overline{0011}$

This is how we convert 0.1 to binary

0.1	*	2	=	**0**.2
0.2	*	2	=	**0**.4
0.4	*	2	=	**0**.8
0.8	*	2	=	**1**.6
0.6	*	2	=	**1**.2
0.2				

At this point, it repeats

float

- Operations on floats may cause **rounding errors**
- This makes it impossible to reason about programs
  - This is why there are no floats in C0
- Adding more bits does not solve the problem
  - The type double of **double-precision** floating point numbers has typically 64 bits nowadays
    - similar issues

Union and Enum Types

Sample Problem

- Print a message based on the season

- How to encode seasons?
  - use strings …
    - testing which season we are in is costly
  - use integers

- Drawbacks
  - The encoding is not mnemonic
    - we will make mistakes
  - A whole int for 4 values seems wasteful

```
// 0 = Winter
// 1 = Spring
// 2 = Summer
// 3 = Fall

int today = 3;
if (today == 0)
  printf("snow!\n");
else if (today == 3)
  printf("leaves!\n");
else
  printf("sun!\n");
```

Enum Types

- *The encoding is not mnemonic*
- *A whole int for 4 values seems wasteful*

- An **enum type** lets
  - the programmer choose mnemonic values
    - no need to remember the encoding – just use the names
  - the compiler decide how to implement them
    - what actual type to map them to
    - what values to use
    - the compiler optimizes space usage

The compiler maps enum names to some numerical values

By convention, enum values are written in all caps

```
enum season { WINTER, SPRING, SUMMER, FALL };

enum season today = FALL;
if (today == WINTER)
  printf("snow!\n");
else if (today == FALL)
  printf("leaves!\n");
else
  printf("sun!\n");
```

Switch Statements

- A **switch** **statement** is an alternative to cascaded if-elses for numerical values
  - ➢ including union types
  - ○ They make the code more readable
- Each value considered is handled by a **case**
  - ○ The execution of a case continues till the next **break** or the end of the switch statement
    - ➢ it exits the switch statement
  - ○ The default case handles any remaining value

```
enum season { WINTER, SPRING, SUMMER, FALL };

enum season today = FALL;

switch (today) {
  case WINTER:
    printf("snow!\n");
    break;

  case FALL:
    printf("leaves!\n");
    break;

  default:
    printf("sun!\n");
}
```

a case

another case

the default case

Switch Statements

- If a break is missing, the execution continues with the next case

This the **source of many bugs**!

```
enum season { WINTER, SPRING, SUMMER, FALL };

enum season today = FALL;

switch (today) {
  case WINTER:
    printf("snow!\n");
    break;

  case FALL:
    printf("leaves!\n");
    break;

  default:
    printf("sun!\n");
}
```

a case

another case

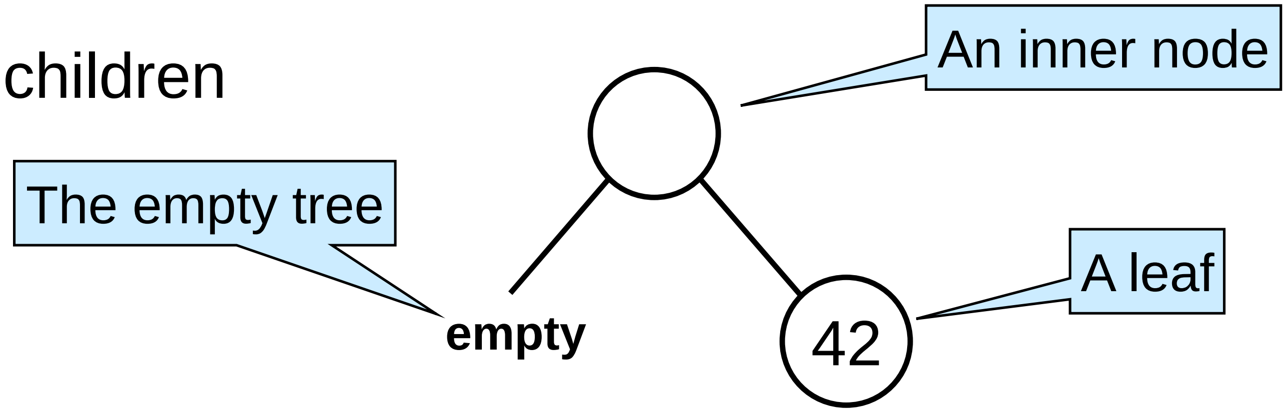
the default case

Recent versions of gcc issue a warning when this happens

Another Sample Problem

- Define a type for binary trees with int data only in their leaves
  - and where the empty tree is **not** represented as NULL

- A **leafy tree** could be
  - an inner node with pointers to two children
  - a leaf with int data
  - an empty tree



- Then:

```
enum nodekind = { INNER, LEAF, EMPTY };

struct ltree {
  enum nodekind kind;
  int data;
  leafytree *left;
  leafytree *right;
};
typedef struct ltree leafytree;
```

We now know about enum types!

Sample Problem

This representation wastes memory

- the compiler will pick a small numerical type for kind ✓
  - probably a char

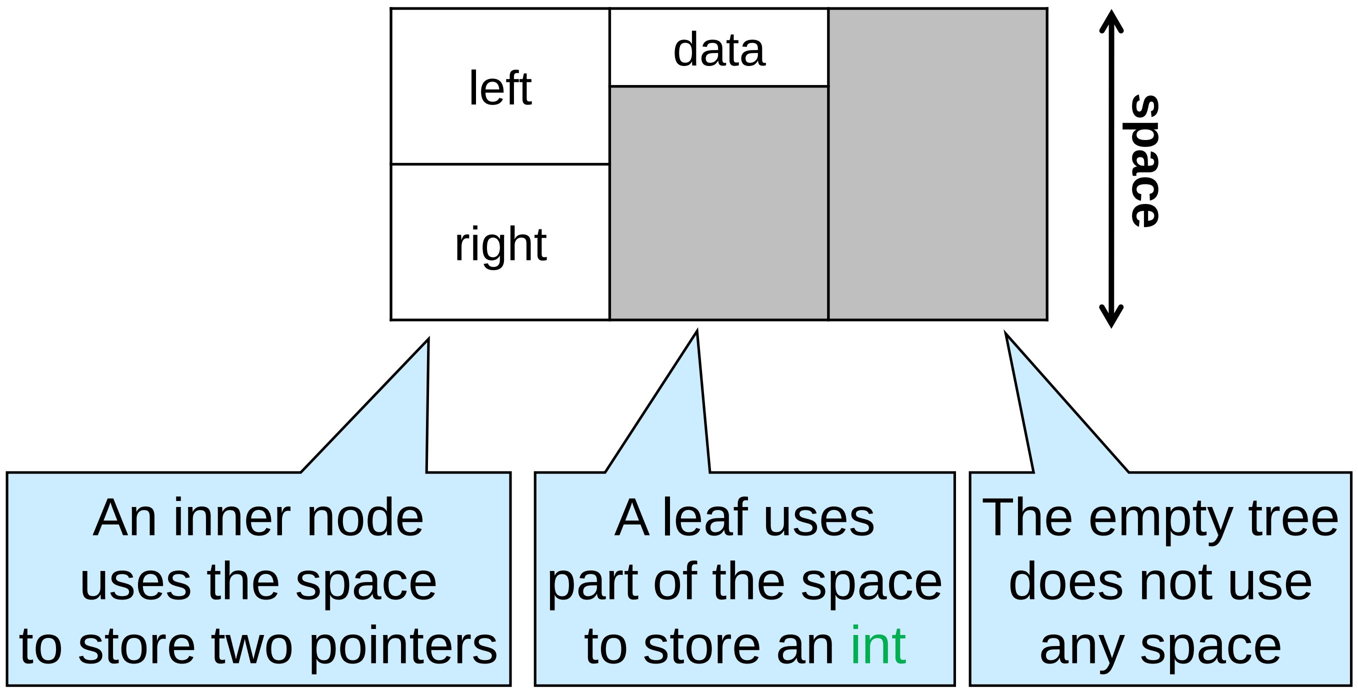
```
enum nodekind = { INNER, LEAF, EMPTY };

struct ltree {
  enum nodekind kind;
  int data;
  leafytree *left;
  leafytree *right;
};
typedef struct ltree leafytree;
```

but

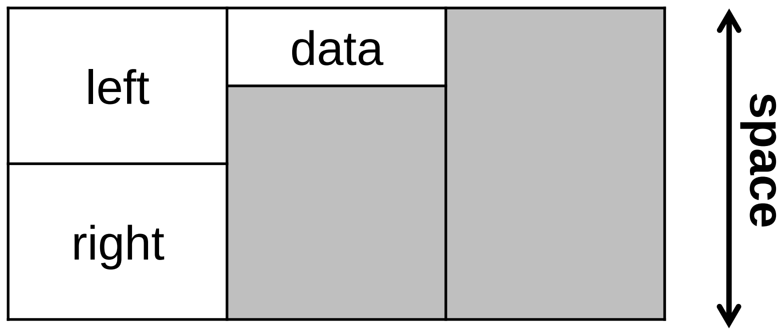
- the remaining 3 fields are never fully utilized for any node type
  - inner nodes do not make use of the data field
  - leaves do not use left and right
  - the empty tree does not need any ✘

Union Types

- A **union type** allows using the same space in different ways
- Consider the space needed for a node, aside from its type



Union Types



- A **union type** allows using the same space in different ways

```
enum nodekind { INNER, LEAF, EMPTY };

struct innernode {
    leafytree *left;
    leafytree *right;
  };

union nodecontent {
    int data;
    struct innernode node;
  };

struct ltree {
    enum nodekind kind;
    union nodecontent content;
};
typedef struct ltree leafytree;
```

An inner node consists of two pointers

The content of a generic node is
- either an int (the data of a leaf)
- or an inner node

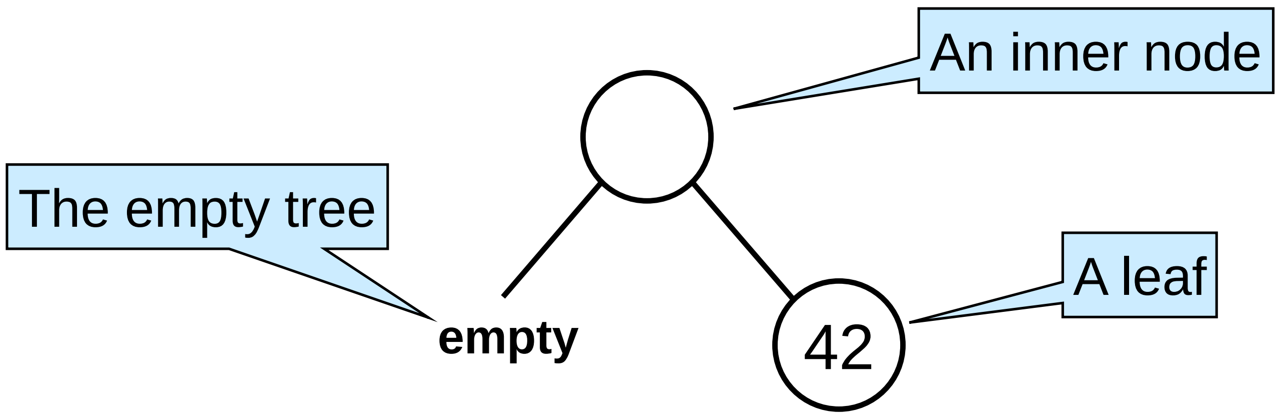
There is no need to have an option for the empty tree since it uses no space

C11 supports a much more compact syntax

Building a Tree

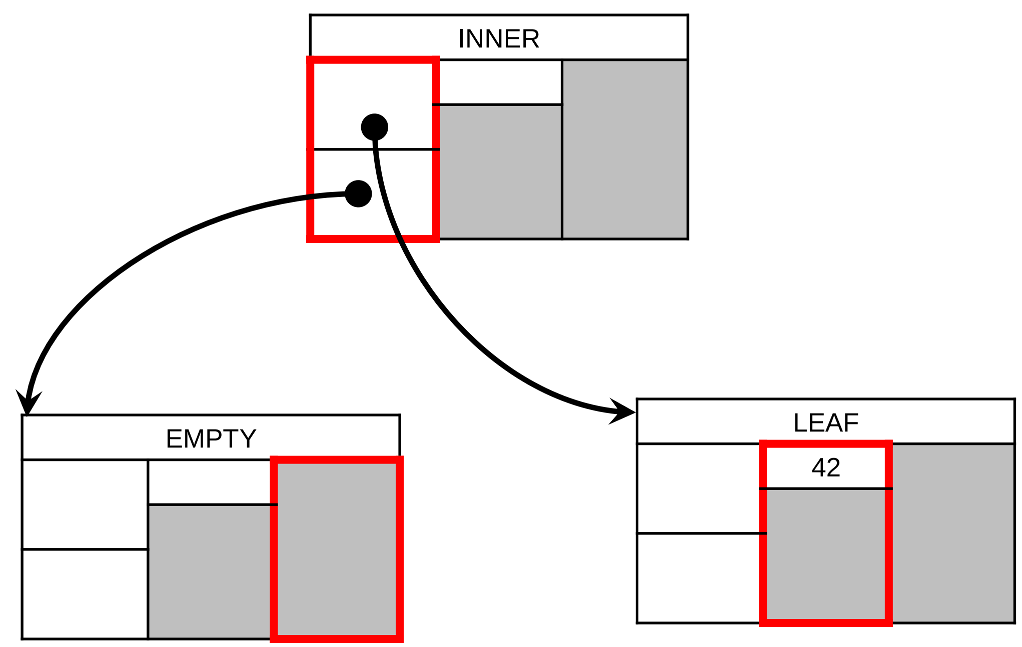
```c
enum nodekind { INNER, LEAF, EMPTY };

struct innernode {
    leafytree *left;
    leafytree *right;
  };

union nodecontent {
    int data;
    struct innernode node;
  };

struct ltree {
    enum nodekind kind;
    union nodecontent content;
};
typedef struct ltree leafytree;
```

- Let’s write code that create this tree




```c
leafytree *T = malloc(sizeof(leafytree));
T->kind = INNER;
T->content.node.left = malloc(sizeof(leafytree));
T->content.node.left->kind = EMPTY;
T->content.node.right = malloc(sizeof(leafytree));
T->content.node.right->kind = LEAF;
T->content.node.right->content.data = 42;
```

Whenever not following a pointer, we must use the dot notation



Adding up a Leafy Tree

- We use a switch statement to write clear code
  - we discriminate on T->kind
  - it has three possible values
    - INNER, LEAF and EMPTY

```
int add_tree(leafytree *T) {
  int n = 0;

  switch (T->kind) {
   case INNER:
     n += add_tree(T->content.node.left);
     n += add_tree(T->content.node.right);
     break;

   case LEAF:
     n = T->content.data;
     break;

   default:
     n = 0;
  }

  return n;
}
```

Summary

Undefined Behavior

Memory	• Reading/writing to non-allocated memory • Reading uninitialized memory • even if correctly allocated • Use after free • Double free • Freeing memory not returned by malloc/calloc • Writing to read-only memory
Numbers	• Division/mod by zero • INT_MIN divided/mod’ed by -1 • Shift by more than the number of bits • Signed overflow