Types in C

LAST

- **C's Memory Model**
  - Arrays and pointers
  - Pointer casting
  - Arrays on the stack
  - Structs on the stack
  - "Address of" operator
- **Undefined behavior**

TODAY

- Numbers in C
- **Implementation-defined behavior**
- Other C types

NEXT

C0 virtual machine

Revisiting last lecture

- Safety violations in C0 is typically undefined behavior in C.
- Pointers and arrays are the same.
- It is not possible to capture or check the length of arrays.

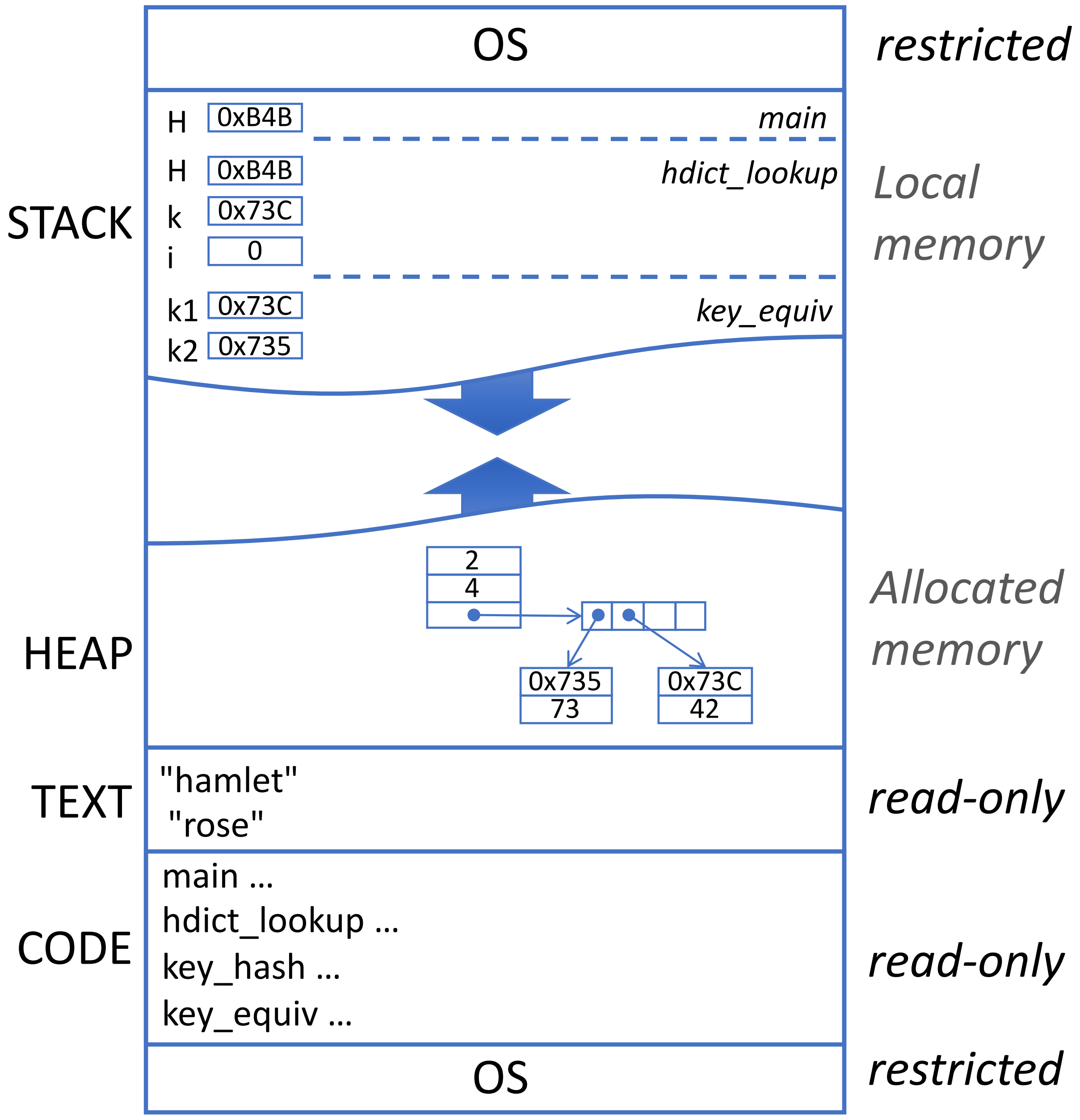
&: "Address of" operator

```
void increment(int *p){
    REQUIRES(p != NULL);
    *p = *p + 1;
}
```

In C, & can be used to get address of any value that has a memory address.

```
int i = 42;
increment(&i);
printf("%d", i);
```

prints 43.



&: "Address of" operator

Allocate a point structure on the stack, initialize the y coordinate and increment it using `increment`.

```
struct point p;
p.x =0;
increment(&p.x);
```

```
void increment(int *p){
    REQUIRES(p != NULL);
    *p = *p + 1;
}
```

```
struct point {
    int x;
    inty;
};
```

Transition to C

LOST	GAINED
Contracts	Preprocessor
Safety	Explicit memory management
Garbage collection	Tools: valgrind
Memory initialization	Pointer arithmetics
Tools: Interpreter (coin)	Stack allocated arrays and structs
Well-behaved arrays	Generalized “address of”
Fully defined language	
Strings	

Size of `int` in C over time

	70s	80s	90s	now
Pointer size	8	16	32	64
`int` size	8	16	32	32

Implementation-defined behavior

Compiler is required to define the size of `int`

- The programmer can find it in <limits.h>

Undefined-behavior for integers

- Division/modulus by 0
- Shift by more than the size of the integer
- Overflow for signed types like `int`

Integer types in C

signed	**unsigned**	**today**	**C99 constraints (signed)**
`signed char`	`unsigned char`	8 bits	exactly 1 byte
`short`	`unsigned short`	16 bits	$(-2^{15}, 2^{15})$
`int`	`unsigned int`	32 bits	$(-2^{15}, 2^{15})$
`long`	`unsigned long`	64 bits	$(-2^{31}, 2^{31})$

Fixed size integers (defined in `<stdint.h>`)

fixed-size signed	**today's signed equivalent**
`int8_t`	`signed char`
`int16_t`	`short`
`int32_t`	`int`
`int64_t`	`long`

Fixed size integers (defined in `<stdint.h>`

fixed-size unsigned	**today's unsigned equivalent**
`uint8_t`	`unsigned char`
`uint16_t`	`unsigned short`
`uint32_t`	`unsigned int`
`uint64_t`	`unsigned long`

`size_t`

- An unsigned integer type
- Preferred way to declare any arguments or variables that hold the size of an object.
- The result of the sizeof operator is of this type, and functions such as malloc accept arguments of this type to specify object sizes. On systems using the GNU C Library, this will be `unsigned int` or `unsigned long int`.

Integer casting

- Literal number always has type `int`
- Changing integer types

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

```
long x = 3;        // Implicitly cast
long y = (long)3; // Explicitly cast
```

Implicit casting is dangerous: `long x = 1 << 40;`

1 is 32 bits and we are shifting it by 40 bits, undefined behavior

Casting rules in C

- When casting signed to/from unsigned numbers of the same size, **bit pattern is preserved.**
- When casting small to big number of same signedness, **value is preserved.**
- When casting big to small number of the same signedness, **make sure the value will fit**. Otherwise undefined behavior.

Casting rules in C

- When casting signed to/from unsigned numbers of the same size, **bit pattern is preserved.**

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

signed char a = -3;                // a is  -3 (0xFD)
unsigned char b = (unsigned char)a;  // b is 253 (0xFD)
```

Casting rules in C

- When casting small to big number of same signedness, **value is preserved.**

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


signed char a = -3;             // a is  -3 (0xFD)
int b = (int)a;                 // b is -3  (0xFFFFFFFD)
```

uses sign extension

Casting rules in C

- When casting signed to/from unsigned numbers of the same size, **bit pattern is preserved.**
- When casting small to big number of same signedness, **value is preserved.**
- When casting big to small number of the same signedness, **make sure the value will fit**. Otherwise undefined behavior.

Casting across both sign and size

```
unsigned char x = 0xF0;  // x is 240
int y = (int)x;
```

```
unsigned char x = 0xF0;
int y = (int)x;
```

Casting across both sign and size

`unsigned char 0xF0 = 240`

cast to `unsigned int`

preserve value

`unsigned int 0x000000F0`

cast to `signed int`

preserve bit pattern

`0x000000F0`
`= 240`

cast to `signed char`

preserve bit pattern

`signed char 0xF0 = –16`

cast to `signed int`

preserve value

`0xFFFFFFF0`
`= –16`

Instead of

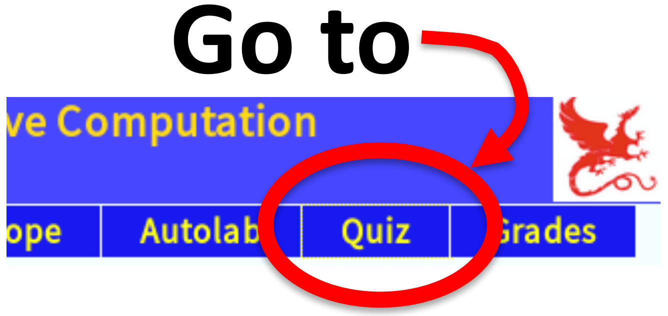
```
unsigned char x = 0xF0;
int y = (int)x;
```

Write the steps explicitly

```
unsigned char x = 0xF0;  // x is 240
int y1 = (int) (unsigned int) x;
printf("y1 is %d\n", y1);
int y2 = (int) (signed char) x;
printf("y2 is %d\n", y2);
```

```
KAYNAR3:code dilsun$ ./a.out
y1 is 240
y2 is -16
```

Go to



or

cs.cmu.edu/~15122/quiz

1. Write a C expression that evaluates to a pointer to the element at index 6 of a 20-element `int` array `A`?
2. In a C executable compiled with `-DDEBUG`, contracts are _comments_macros_removed_executed_undefined_
3. **True/False:** C allows allocating structs on the stack.
4. What program do we use to find out-of-bound array accesses in C code ? _gcc__cc0__valgrind__cpp__ls__
5. **True/False**: All safety violations in C0/C1 are undefined behaviors in C.

Floating point numbers `<float.h>`

double precision

```
float x = 0.1;                  double x = 0.1;
float y = 2.0235E27;            double y = 2.0235E27;
```

```
(10E20 / 10E10) * 10E10 != 10E20;
```

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

infinite loop!

Enumarations

```
int WINTER = 0;
int SPRING = 1;
int SUMMER = 2;
int FALL = 3;

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

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


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

Switch statements

Replacing if/else if/…/else if/else with switch

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

enum season_type season = FALL;
switch (season) {
  case WINTER:
    printf("snow!\n");
    break;
  case FALL:
    printf("leaves!\n");
    break;
  default:
    printf("sun!\n");
}
```

Transition to C

LOST	GAINED
Contracts	Preprocessor
Safety	Explicit memory management
Garbage collection	Tools: valgrind
Memory initialization	Pointer arithmetics
Tools: Interpreter (coin)	Stack allocated arrays and structs
Well-behaved arrays	Generalized “address of”
Fully defined language	More numerical types
Strings	