printf

Like C0, C provides printf to print values to terminal. However, C supports many more format specifiers than C0 (which has only %d, %s and %c). Particularly useful are

- %u to print an unsigned int,
- %ld to print a long,
- %lu to print an unsigned long, and
- %zu to print a size_t, and

Feel free to search online for format specifiers for more types[^1]

An argument corresponding to %d (or %i) must have type int (or smaller signed types like short and signed char). Providing an argument of any other type is undefined behavior — it may print the expected result, or it may not on any given execution. Thus,

```c
int z = -500;
printf("%u\n", z);
```

is undefined behavior.

structs on the stack

In C0 and C1, if we ever wanted to create a struct, we had to explicitly allocate memory for it using alloc. C doesn’t have this restriction — you can declare struct variables on the stack, just like int’s. We set a field of a struct with dot-notation, below. Recall that when we had a pointer p to a struct, we accessed its fields with p->data. This is just syntactic sugar for (*p).data.

Checkpoint 0

```c
#include <stdio.h>

struct point {
    int x;
    char y;
};

int main () {
    struct point a;
    a.x = 3;
    a.y = 'c';
    struct point b = a;
    b.x = 4;
    printf("a.x, a.y: %d, %c\n", a.x, a.y); // what gets printed out here?
    printf("b.x, b.y: %d, %c\n", b.x, b.y); // how about here?
}
```

Addressing all things

We have already seen the “address-of” operator, &, used to find function pointers in C1. In C, we can do the same thing with variables. This is useful if you want to give a function a reference to a local variable. *Remember to only free pointers returned from malloc!*

Checkpoint 1

```c
#include <stdio.h>
#include "lib/contracts.h"

void bad_mult_by_2(int x) {
    x = x * 2;
}

void mult_by_2(int* x) {
    REQUIRES(x != NULL);
    *x = *x * 2;
}

int main () {
    int a = 4;
    int b = 4;
    bad_mult_by_2(a);
    mult_by_2(&b);
    printf("a: %d  b: %d\n", a, b);
    return 0;
}
```

```c
#include <stdio.h>
#include "lib/contracts.h"

struct point {
    int x;
    int y;
};
void swap_points(struct point* P) {
    REQUIRES(P != NULL);
    int temp = P->x;
    P->x = P->y;
    P->y = temp;
}

int main() {
    struct point A;
    A.x = 122;
    A.y = 15;
    swap_points(&A);
    printf("A: (%d, %d)\n", A.x, A.y);
    return 0;
}
```

What is the output when each of these programs are run?
Casting

C provides many different types to represent integer values. Some are signed while other are unsigned, and they don’t necessarily are 32-bit long (for example a short is commonly 16 bits).

Sometimes, if we really know what we are doing, we may want or need to convert between these types. We can do so by casting. The flow chart to the right summarizes what happens when casting a numerical expression $exp$ of type $old\_type$ to type $new\_type$.

The general rule of thumb is that value is preserved whenever possible, and the bit pattern is preserved otherwise.

Here is one example of each situation:

// -3 is representable as an int
signed char x = -3; // x is -3 (= 0xFD)
int y = (int)x; // y is -3 (= 0xFFFFFFFF)

// -241 is NOT representable as a SIGNED char and the new type is signed
int x = -241; // x is -241(= 0xFFFFFFFFF)
signed char y = (signed char)x; // y is ?? (often 0x0F)

// -3 is NOT representable as a UNSIGNED int, the new type is bigger
signed char x = -3; // x is -3 (= 0xFD)
unsigned int y = (unsigned int)x; // y is 4294967293 (= 0xFFFFFFFF)

// -3 is NOT representable as a UNSIGNED char, the new type and smaller or equal
signed char x = -3; // x is -3 (= 0xFD)
unsigned char y = (unsigned char)x; // y is 253 (= 0xFD)

Most casts between pointers and integers are implementation-defined.
switch statements

A switch statement is a different way of expressing a conditional. Here’s an example:

```c
void print_dir(char c) {
    switch (c) {
    case 'l':
        printf("Left
");
        break;
    case 'r':
        printf("Right
");
        break;
    case 'u':
        printf("Up
");
        break;
    case 'd':
        printf("Down
");
        break;
    default:
        fprintf(stderr, "Specify a valid direction!\n");
    }
}
```

Each case’s value should evaluate to a constant integer type (this can be of any size, so chars, ints, long long ints, etc).

The break statements here are important: If we don’t have them, we get fall-through: without the break on line 11 we’d print “Up” and then “Down” for case ‘u’.

**Checkpoint 2**

Fall-through is useful but can be tricky. What’s wrong with the following code? How do you fix it?

```c
#include <stdio.h>
#include <stdlib.h>
void check_parity(int x) {
    switch (x % 2) {
    case 0:
        printf("x is even!\n");
    default:
        printf("x is odd!\n");
    }
}
```
Checkpoint 3
What’s wrong with each of these pieces of code?

(a) 1 int* add_sorta_maybe(int a, int b) {
  2     int x = a + b;
  3     return &x;
  4 }

(b) 1 void print_int(int* i) {
  2     printf("%d\n", *i);
  3     free(i);
  4 }

  5  
  6  
  7  int main() {
  8     int x = 6;
  9     print_int(&x);
 10     return 0;
 11 }

(c) 1 int main() {
  2     int x = 0;
  3     if (x = 1)
  4         printf("woo\n");
  5     return x;
  6 }

(d) 1 int main () {
  2     unsigned int x = 0xFE1D;
  3     short y = (short)x;
  4     return 0;
  5 }

(e) 1 int main() {
  2     char* s = "15-122";
  3     s[4] = '1'; // blasphemy
  4     printf(s);
  5     return 0;
  6 }

(f) 1 int main() {
  2     char s[] = {'a', 'b', 'c'};
  3     printf("%s\n", s);
  4     return 0;
  5 }