15-122: Principles of Imperative Computation Spring 2024

Recitation 11: C-ing is Believing Thursday March 28th

`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`.

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,

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

is undefined behavior. See the *Guide to Success on Printing in C* for more information about `printf`.

`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`.

---

[1] The C++ document `http://cplusplus.com/reference/cstdio/printf` is a good reference (C behaves similarly).

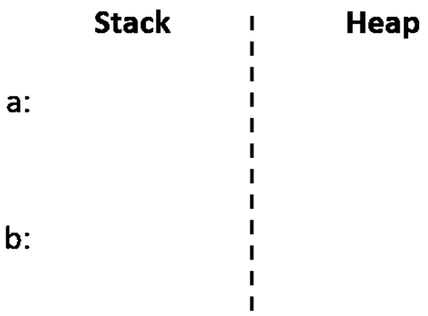
Checkpoint 0

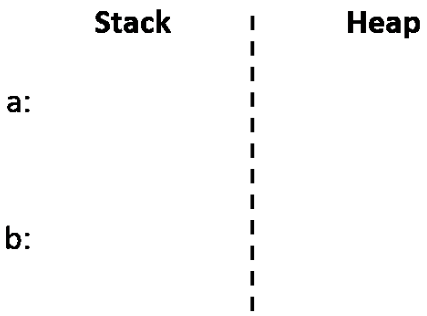
Here are two programs that are identical except that one allocates a **`struct`** `point` on the stack and the other on the heap. Write down what the two pairs of `printf` statements will print. You may want to trace both programs using the memory diagram templates below.

```
#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;
  b.y = ’d’;

  // what gets printed out here?
  printf("a.x, a.y: %d, %c\n", a.x, a.y);
  // how about here?
  printf("b.x, b.y: %d, %c\n", b.x, b.y);

}
```

```
int main () {
  struct point∗ a = xmalloc(sizeof(struct point));
  a->x = 3;
  a->y = ’c’;
  struct point∗ b = a;
  b->x = 4;
  b->y = ’d’;

  // what gets printed out here?
  printf("a->x, a->y: %d, %c\n", a->x, a->y);
  // how about here?
  printf("b->x, b->y: %d, %c\n", b->x, b->y);
  free(a);
}
```





Addressing all things

We have already seen the "address-of" operator, `&`, used to get 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` *or* `calloc`*!*

Checkpoint 1

```
#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;
}
```

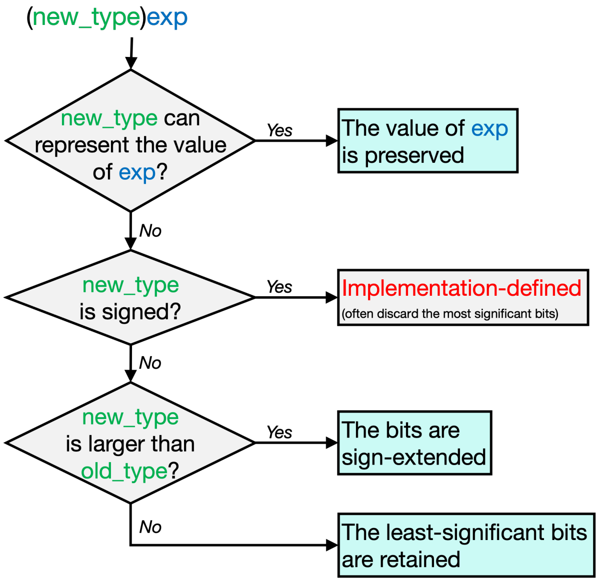
What is the output when this program is run?

Casting

C provides many different types to represent integer values. Some are signed while others are unsigned, and they aren't necessarily 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 (= 0xFFFFFFFD)

// -241 is NOT representable as a SIGNED char and the new type is signed
int x = -241;                        // x is -241(= 0xFFFFFF0F)
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 (= 0xFFFFFFFD)

// -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)
```

Checkpoint 2

Assume that a **`char`** is 8 bits and an **`int`** is 32 bits and that negative numbers use two's complement.

- The values represented by an **`int`** range from `-2147483648` to `2147483647`.
- The values represented by an **`unsigned int`** range from ______________ to ______________.
- The values represented by a **`signed char`** range from ______________ to ______________.
- The values represented by an **`unsigned char`** range from ______________ to ______________.

`switch` statements

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

```c
 1 void print_dir(char c) {
 2   switch (c) {
 3     case 'l':
 4       printf("Left\n");
 5       break;
 6     case 'r':
 7       printf("Right\n");
 8       break;
 9     case 'u':
10       printf("Up\n");
11       break;
12     case 'd':
13       printf("Down\n");
14       break;
15     default:
16       fprintf(stderr, "Specify a valid direction!\n");
17   }
18 }
```

Each case's value should evaluate to a constant integer type (this can be of any size, so **`char`**s, **`int`**s, **`long long int`**s, 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 3

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");
  }
}
```

Common Pitfalls

Checkpoint 4

What’s wrong with each of these pieces of code?

(a)

```c
int* add_sorta_maybe(int a, int b) {
  int x = a + b;
  return &x;
}
```

(b)

```c
void print_int(int* i) {
  printf("%d\n", *i);
  free(i);
}

int main() {
  int x = 6;
  print_int(&x);
  return 0;
}
```

(c)

```c
int main() {
  int *A[2];
// A stack-allocated 2-element int* array
  A[0] = xmalloc(sizeof(int));
  A[1] = A[0];
  free(A[0]);
  free(A[1]);
  return 0;
}
```

(d)

```c
int main () {
  unsigned int x = 0xFE1D;
  short y = (short)x;
  return 0;
}
```

(e)

```c
int main() {
  char* s = "15-122";
  s[4] = ’1’; // blasphemy
  printf(s);
  return 0;
}
```

(f)

```c
int main() {
  char s[] = {’a’, ’b’, ’c’};
  printf("%s\n", s);
  return 0;
}
```