15-122: Principles of Imperative Computation

R11: C-ing is Believing

Josh, Andrew, Aaron

printf

In C0 and C1, we had different print functions for each type i.e. `printint`, `println`, etc (these were included in `<conio>`). In C, there is just one main print function: `printf` (included in `<stdio.h>`). `printf` always takes in a string, but you can embed "format specifiers" to print other types.

Checkpoint 0

```
1 #include <stdio.h>
2
3 int main()
4 {
5   int x = 64;
6   printf("%d\n", x); // decimal (can also use %i)
7   printf("%x\n", x); // hexadecimal
8   char a = 'd'; // single quotes!
9   printf("%c\n", a);
10  printf("%d\n", a);
11  char* s = "hello"; // double quotes!
12  printf("%s\n", s);
13  void* p = (void*)0xdeadbeef;
14  printf("%p\n", p); // must be a void*, not any other pointer type
15  size_t z = 8;
16  printf("%zu\n", z);
17 }
```

What is the output of this program?

structs on the stack

In C0 and C1, if we ever wanted to create a `struct`, we had to explicitly allocate memory for it via a call to `alloc`. C doesn't have this restriction - if you like, you can declare `struct` variables on the stack, just like `int`'s. We set a field of a `struct` with dot-notation, as follows:

Checkpoint 1

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct point {
5    int x;
6    char y;
7 };
8 int main () {
9    struct point a;
10   a.x = 3;
11   a.y = 'c';
12   struct point b = a;
13   printf("a.x, a.y: %d, %d\n", a.x, a.y); // what gets printed out here?
14   printf("b.x, b.y: %d, %c\n", b.x, b.y); // how about here?
15 }
```

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`.

Addressing all the 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 2

```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 introduces many different types to represent integer values. 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*.

(a) Casting from small signed to large signed: the value is sign-extended with the sign bit (most significant bit) but remains unchanged

(b) Casting from small unsigned to large unsigned: the value is padded with leading zeroes but remains unchanged

(c) Casting from signed to unsigned of the same size: the value in the new type is equal to that in the old type, modulo `INT_MAX` for the new type. In other words, the bit pattern does not change, but if the original value was negative, then the value will change.

(d) Casting from unsigned to signed of the same size: if the unsigned value is expressible in the signed type, then that is the new value and neither the bit pattern nor the value changes. Otherwise this is implementation-defined.

Casts between pointers and integers, or casts between large integer types to small integer types are both implementation-defined. Additionally, we don't require you to know what happens when casting directly from small to large between signed and unsigned.[1]

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

[1] It is either implementation defined, or confusing. See section 6.3.1.3 of the C99 standard at `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`.

`switch` statements

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

```
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          break;
18    }
19 }
```

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'`.

Here's some code that takes a positive number at most 10 and determines whether it is a perfect square. The behavior here is called fall-through.

```
1  int is_perfect_square(int x) {
2     REQUIRES(1 <= x && x <= 10);
3     switch (x) {
4        case 1:
5        case 4:
6        case 9:
7           return 1;
8           break;
9        default:
10          return 0;
11          break;
12    }
13 }
```

Fall-through is useful, but can be tricky. What's wrong with the following code, and how do you fix it?

Checkpoint 3

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  void check_parity(int x) {
4    switch (x % 2) {
5     case 0:
6       printf("x is even!\n");
7     default:
8       printf("x is odd!\n");
9    }
10 }
```

Checkpoint 4

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

c)

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

d)

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

e)

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

f)

```
1 void print_int(int* i) {
2    printf("%d\n", *i);
3    free(i);
4 }
5
6 int main() {
7    int x = 6;
8    print_int(&x);
9    return 0;
10 }
```