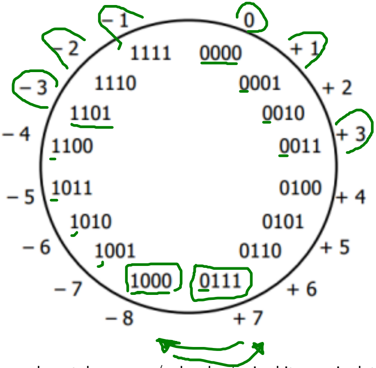
Number representations: integers

C integers use "two's complement" representation for signed integers. Illustration with 4 bits:

Two's complement system can cover both negative and positive values:
1) the leftmost bit can tell you If the number is positive or negative
2) take binary form of any positive number in two's complement system, flip all its bits using negation (i.e., "~") operator, add 1 to it, it will give you the corresponding negative value in binary form
3) if you add 1 to the last positive value (i.e., maximum value), it will wrap around and will give the last negative value (i.e., minimum value)



http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php

$2^4 = 16$

$\sim(0011) \Rightarrow 1100 + 1 \Rightarrow 1101$

Integer overflow

When a two's complement number overflows, it wraps around to a negative number

overflow.c:

```
#include <stdio.h>

int main() {
    int i = 2147483647; // largest integer value
    int i_plus_1 = i + 1;
    printf("i = %d, i+1 = %d\n", i, i_plus_1);
    return 0;
}
```

will overflow and will wrap around since it is a two's complement representation

```
$ gcc -std=c99 -pedantic -Wall -Wextra -c overflow.c
$ gcc -o overflow overflow.o
$ ./overflow
i = 2147483647, i+1 = -2147483648
```

Number representation differences

Integer and floating-point representations differ:

- Integers have limited range, but integers in the range can be represented precisely. Floating point have limited range and can only approximate most numbers in the range.
- Integers use all available bits for two's-complement representation. Floating point have separate sets of bits for sign, exponent and mantissa.

Conversions between types

- When you do `float a = 1` or `int i = 3.0`, it's not as simple as copying bits
- When going from integer types to to `float` (or `double`), we are getting an approximation, not the exact integer
- Our CSF (Computer System Fundamentals) course goes into detail on conversions and arithmetic involving these representations

Integer literal types

- The type for an *integer literal* (e.g. 88, −1000000000) is determined based on its value
  - Specifically: its type is the smallest integer type that can store it without overflowing

Integer literal types

int_literal.c:

```c
#include <stdio.h>

int main() {
    int a = 1;
    int b = -3000;
    long c = 10000000000; // too big for an int
    printf("%d, %d, %ld\n", a, b, c);
    return 0;
}
```

```
$ gcc -std=c99 -pedantic -Wall -Wextra -c int_literal.c
$ gcc -o int_literal int_literal.o
$ ./int_literal
1, -3000, 10000000000
```

Integer literal range

Compiler can warn us when an integer literal is too big for a particular variable

```
int_literal2.c:
#include <stdio.h>

int main() {
    int a = 10000000000;  // too big for an int
    // a will "overflow" and "wrap around" to some other number
    printf("%d\n", a);
    return 0;
}
```

```
$ gcc -std=c99 -pedantic -Wall -Wextra -c int_literal2.c
int_literal2.c: In function ‘main’:
int_literal2.c:4:13: warning: overflow in conversion from ‘long int’ to ‘int’ c
    4 |     int a = 10000000000;  // too big for an int
      |             ^~~~~~~~~~~
$ gcc -o int_literal2 int_literal2.o
$ ./int_literal2
1410065408
```

Floating-point literal types

Floating-point literal (e.g. `3.14`, `-.7`, `-1.1e-12`) has type `double`

You can force it to be `float` by adding `f` suffix

float_literal.c:

specifying that the floating-point is a float and not a double

double

```c
#include <stdio.h>

int main() {
    float a = 3.14f;
    double b = 33.33, c = -1.1e-12;
    printf("%f, %f, %e\n", a, b, c);
    return 0;
}
```

-1.1×10^{-12}

```
$ gcc -std=c99 -pedantic -Wall -Wextra -c float_literal.c
$ gcc -o float_literal float_literal.o
$ ./float_literal
3.140000, 33.330000, -1.100000e-12
```

Floating-point literal range

Again, compiler will warn if literal doesn't fit

float_literal2.c:

```c
#include <stdio.h>

int main() {
    float a =  2e128f; // too big; max float exponent is 127
    double b = 2e1024; // too big; max double exponent is 1023
    printf("%f, %f\n", a, b);
    return 0;
}
```

Floating-point literal range

```
$ gcc -std=c99 -pedantic -Wall -Wextra -c float_literal2.c
float_literal2.c: In function ‘main’:
float_literal2.c:4:5: warning: floating constant exceeds range of ‘floa
    4 |     float a =  2e128f; // too big; max float exponent is 127
      |     ^~~~~~
float_literal2.c:5:5: warning: floating constant exceeds range of ‘doub
    5 |     double b = 2e1024; // too big; max double exponent is 1023
      |     ^~~~~~
$ gcc -o float_literal2 float_literal2.o
$ ./float_literal2
inf, inf
```

Type Conversions

C can automatically convert between types "behind the scenes"

This is called *automatic conversion* and can be a ***promotion*** in the case of a smaller type value converted to a larger type, or ***narrowing*** in the case of a larger type value converted to a smaller type.

```
float ten = 10;
// float <- int

int ten = 10.585; // truncates to 10
// int <- float
```

double

Promotion

promotion_1.c:

```c
#include <stdio.h>

int main() {
    int a = 1;
    float f = a * 1.5f;
    printf("%f\n", f);
    return 0;
}
```

```
$ gcc -std=c99 -pedantic -Wall -Wextra -c promotion_1.c
$ gcc -o promotion_1 promotion_1.o
$ ./promotion_1
1.500000
```

Promotion

```
int a = 1;
float f = a * 1.5f;
```

Note operands types: `a` is `int`, `1.5f` is `float`

When operand types don't match, "smaller" type is promoted to "larger" before applying operator

`char` < `int` < `unsigned` < `long` < `float` < `double`

promotion

narrowing

Promotion

```
promotion_2.c:
#include <stdio.h>

int main() {
    int a = 3;
    float f = a / 2;
    printf("%f\n", f);
    return 0;
}
```

```
$ gcc -std=c99 -pedantic -Wall -Wextra -c promotion_2.c
$ gcc -o promotion_2 promotion_2.o
$ ./promotion_2
1.000000
```

Promotion

```
int a = 3;
float f = a / 2;
```

No promotion here before division, since operands `a` and `2` are same type: `int`

Narrowing

Type conversions from larger to smaller types (e.g. `double` -> `float` or `long` -> `int`) can happen automatically too - explicit type casts are not required!

Sometimes called *narrowing* conversions, these usually chop off the extra bits without rounding values.

narrow_1.c:

```
#include <stdio.h>

int main() {
    unsigned long a = 1000;
    int b = a; // automatic *narrowing* conversion
    float c = 3.14f;
    double d = c; // automatic conversion
    printf("b=%d, d=%f\n", b, d);
    return 0;
}
```

Narrowing

```
$ gcc -std=c99 -pedantic -Wall -Wextra -c narrow_1.c
$ gcc -o narrow_1 narrow_1.o
$ ./narrow_1
b=1000, d=3.140000
```

No compiler warnings.

Narrowing

narrow_2.c:

```
#include <stdio.h>
int square(int num) {
    return num * num;
}
int main() {
    printf("square(2.5)=%d\n", square(2.5));
    return 0;
}
```

Narrowing

```
$ gcc -std=c99 -pedantic -Wall -Wextra -c narrow_2.c
$ gcc -o narrow_2 narrow_2.o
$ ./narrow_2
square(2.5)=4
```

2.5 becomes 2 when passed to `square`. Again no compiler warning.

Narrowing

A value's type is narrowed *automatically* and *without a compiler warning* when: (a) assigning to a variable of narrower type, and (b) passing an argument into a parameter of narrower type.

Other automatic narrowing situations typically yield compiler warnings - you should eliminate the warnings in your programs by using explicit type casts!

Narrowing

casting_1.c:

size_t

```
#include <stdio.h>

int main() {
    printf("sizeof(long)=%d\n", sizeof(long));
    return 0;
}
```

```
$ gcc -std=c99 -pedantic -Wall -Wextra -c casting_1.c
casting_1.c: In function ‘main’:
casting_1.c:4:27: warning: format ‘%d’ expects argument of type ‘int’,
    4 |     printf("sizeof(long)=%d\n", sizeof(long));
      |                          ~^     ~~~~~~~~~~~~
      |                           |     |
      |                          int    long unsigned int
      |                          %ld
$ gcc -o casting_1 casting_1.o
$ ./casting_1
sizeof(long)=8
```

Casting ≈ manual conversion

Some types just can't be used for certain things. E.g. a `float` can't be an array index:

```
casting_2.c:
#include <stdio.h>

int main() {
    int array[] = {2, 4, 6, 8};
    float f = 3.0f;
    printf("array[0]=%d, array[%f]=%d\n", array[0], f, array[f]);
    return 0;
}
```

Casting

```
$ gcc -std=c99 -pedantic -Wall -Wextra -c casting_2.c
casting_2.c: In function ‘main’:
casting_2.c:6:61: error: array subscript is not an integer
    6 |     printf("array[0]=%d, array[%f]=%d\n", array[0], f, array[f]
      |                                                             ^
```

Casting

Type *casting* gives you more control over when promotion and narrowing happens in your program

Casting is sometimes the only way to avoid compiler errors and warnings

Even when conversion would happen automatically, making it explicit by using a cast can make your code clearer

Casting is a higher precedence operation than the binary arithmetic operators since it is a unary operation

Casting

```
casting_3.c:
#include <stdio.h>

int main() {
    int a = 3;
    float f = (float) a / 2;
    //        ^^^^^^^
    // a gets *cast* to float
    // 2 gets *promoted* to float before division
    printf("%f\n", f);
    return 0;
}

$ gcc -std=c99 -pedantic -Wall -Wextra -c casting_3.c
$ gcc -o casting_3 casting_3.o
$ ./casting_3
1.500000
```

Casting

```
casting_4.c:
#include <stdio.h>

int main() {
    printf("sizeof(long)=%d\n", (int)sizeof(long));
    //                           ^^^^^
    return 0;
}
```

```
$ gcc -std=c99 -pedantic -Wall -Wextra -c casting_4.c
$ gcc -o casting_4 casting_4.o
$ ./casting_4
sizeof(long)=8
```

Casting

```
casting_5.c:
#include <stdio.h>

int main() {
    int array[] = {2, 4, 6, 8};
    float f = 3.0f;
    printf("array[0]=%d, array[%d]=%d\n",
           array[0], (int)f, array[(int)f]);
    //                ^^^^^          ^^^^^
    return 0;
}
```

```
$ gcc -std=c99 -pedantic -Wall -Wextra -c casting_5.c
$ gcc -o casting_5 casting_5.o -lm
$ ./casting_5
array[0]=2, array[3]=8
```

Type mystery

pow(x, y) $\Rightarrow x^y$

```
casting_6.c:
#include <stdio.h>
#include <math.h>

int main() {
    float p = 2000.0, r = 0.10;
    float ci_1 = p * pow(1 + r, 10);        1+1+1 +1 +1= 5
    float ci_2 = p * pow(1.0f + r, 10);     1+1+1+1 = 4
    float ci_3 = p * pow(1.0 + r, 10);      1+1+1+1=4
    printf("%.3f\n%.3f\n%.3f\n", ci_1, ci_2, ci_3);
    return 0;
}

$ gcc -std=c99 -pedantic -Wall -Wextra -c casting_6.c
$ gcc -o casting_6 casting_6.o -lm
$ ./casting_6
5187.486
5187.486
5187.485
```

Type mystery: solved

Prototype for `pow`: `double pow(double, double);`

Type promotions are happening both because of the addition and because of the call to `pow`

- `ci_1`: `1` converted to `float`, then added to `r`, then result is converted to `double`
- `ci_2`: `1.0f + r` converted to `double`
- `ci_3`: `r` converted to `double`, then added to `1.0` (already a `double`)

`(float)1` and `(double)1` are not the same. `ci_1` and `ci_2` use `(float)1`, `ci_3` uses `(double)1`.