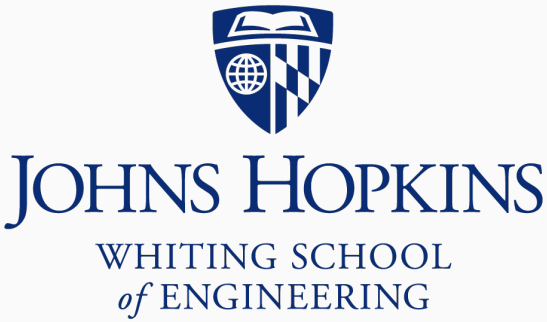
Casting, promotion and narrowing

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Casting, promotion and narrowing

The type for an *integer literal* (e.g. `88`, `-1000000000`) is determined based on its value

Specifically: the smallest integer type that can store it without overflowing

Casting & numeric types

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

Casting & numeric types

Compiler can warn us when an integer literal is too big

```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 -c int_literal2.c -std=c99 -pedantic -Wall -Wextra
int_literal2.c: In function 'main':
int_literal2.c:4:13: warning: overflow in implicit constant conversion
[-Woverflow]
     int a = 10000000000;  // too big for an int
             ^~~~~~~~~~
$ gcc -o int_literal2 int_literal2.o
$ ./int_literal2
1410065408
```

Casting & numeric 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

```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;
}
```

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

Casting & numeric types

Again, compiler will warn if literal doesn't fit

```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;
}
```

Casting & numeric types

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

Promotion

C can automatically convert between types “behind the scenes”

This is called *promotion* or *automatic conversion*

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

Promotion

```c
#include <stdio.h>

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

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

$$\texttt{char} < \texttt{int} < \texttt{unsigned} < \texttt{long} < \texttt{float} < \texttt{double}$$

Promotion

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

E.g. `1 + 1.0`: `int` `1` is converted *directly* to `double` before addition; types "in between" (`unsigned`, `long`, `float`) aren't involved

Promotion

```c
#include <stdio.h>

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

Promotion

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

Promotion

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

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

Narrowing

Type conversions that are *not* promotions (e.g. `double` -> `float` or `long` -> `int`) can happen automatically too

Sometimes called *narrowing* conversions

```
#include <stdio.h>

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

Narrowing

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

No warnings

Narrowing

```c
#include <stdio.h>

int square(int num) {
    return num * num;
}

int main() {
    printf("square(2.5)=%d\n", square(2.5));
    return 0;
}
```

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

2.5 becomes 2 when passed to square. 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

Narrowing

```c
#include <stdio.h>

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

```
$ gcc -c casting_1.c -std=c99 -pedantic -Wall -Wextra
casting_1.c: In function 'main':
casting_1.c:4:27: warning: format '%d' expects argument of type 'int', but
argument 2 has type 'long unsigned int' [-Wformat=]
     printf("sizeof(long)=%d\n", sizeof(long));
                          ~^
                          %ld
$ gcc -o casting_1 casting_1.o
$ ./casting_1
sizeof(long)=8
```

Casting

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

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

Casting

Casting gives you control over when promotion or narrowing happen in your program

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

Even when conversion would happen automatically, making it explicit with casting can make your code clearer

Casting

```c
#include <stdio.h>

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

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

Casting

```c
#include <stdio.h>

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

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

Casting

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

Casting

Pointers can get automatically converted too:

```
int *buf = malloc(400 * sizeof(int));
//       ^^^ void * -> int *

fread(buf, sizeof(int), 400, file);
//    ^^^ int * -> void *
```

Casting

You can cast pointers. Cast might be necessary to avoid a warning:

```
#include <stdio.h>

int main() {
    int n = 40;
    printf("address %p = %d\n", (void*)&n, n);
    //                          ^^^^^^^
    return 0;
}
```

```
$ gcc -c ptr_eg3.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o ptr_eg3 ptr_eg3.o
$ ./ptr_eg3
address 0x7ffd11443e2c = 40
```

Casting

Different types are represented differently in memory, so casting from one type of pointer to the other is almost never OK:

```
#include <stdio.h>

int main() {
    int hello[] = {'h', 'e', 'l', 'l', 'o', '\0'};
    printf("%s\n", (char*)hello);
    return 0;
}
```

```
$ gcc -c ptr_cast.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o ptr_cast ptr_cast.o
$ ./ptr_cast
h
```

Type mystery

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

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

```
$ gcc -c casting_6.c -std=c99 -pedantic -Wall -Wextra
$ 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`.