Modifying Data Sizes (Casting)

ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Casting

- Statically typed programming languages (C/C++, Java, Rust, Go, etc.) all provide “casting” to convert a value from one type for another
  - Other languages automatically adjust data size/type based on values, e.g., Python, which is both good and bad
- Casting has many uses and is often necessary
  - An external library produces values of some type, and you want to pass them to another external library that expects values of some other type
  - Some method returns a 4-byte integer, but from application logic you know that the value is necessarily between 61 and 80, and you want to pass it as a 1-byte ASCII code to some other method
- Casting can be problematic
  - If you need to cast values all the time in your program, perhaps your design/approach is flawed
  - Casting a lot also prevents the compiler from detecting type errors
  - If you don’t know what you’re doing you will break things because casts can lead to wrong (numerical) results

Casting Integers

- Casting can be implicit or explicit (see next slides)
- Casting can change
  - The size of a value (remove / add bits)
  - The interpretation of a value
- In these lectures notes we only consider casts of values interpreted as integers into values also interpreted as integers
  - To make values smaller / bigger
  - To make signed / unsigned values unsigned / signed
- Let's start with making values smaller (in number of bits), which is sometimes called *type narrowing*

High-Level Type Narrowing

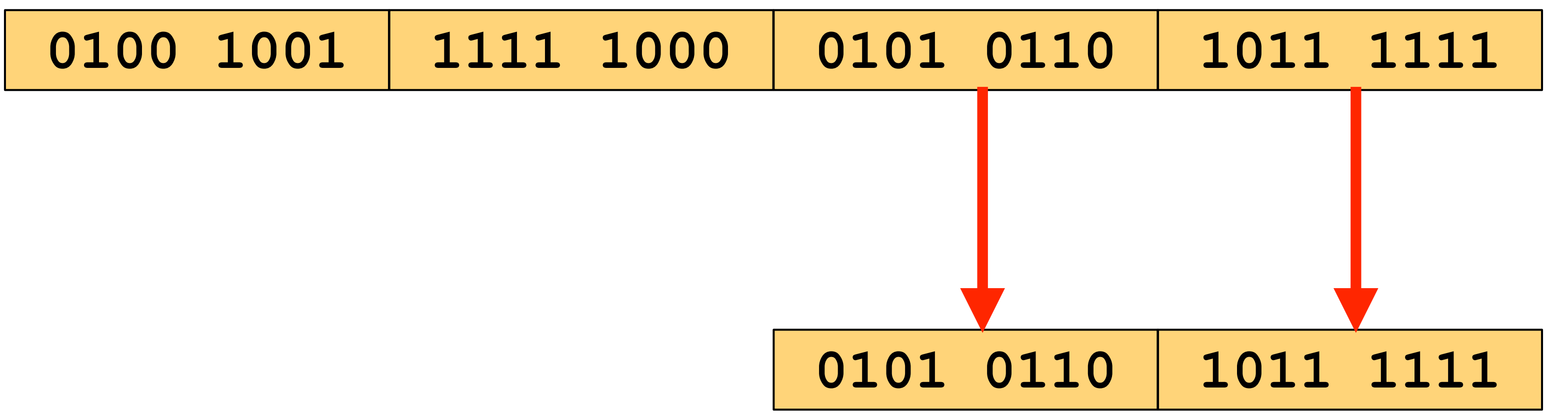
```
int a = 65535;        // 4-byte
short b = a;          // Implicit cast to a 2-byte value
                      // no compiler error/warning
printf("%d\n",b);     // prints ????

int x = -50000;
short y = x;
printf("%d\n",y);     // prints ????
```

```
int a = 65535;
short b = a; // Implicit cast
             // Compiler error: "incompatible types:
             // possible lossy conversion from int to short"

short c = (short)a;     // Explicit cast: no error/warning
System.out.println(c);  // prints ???

int x = -50000;
short y = (short)x;
System.out.println(y);  // prints ???
```

- The goal will be to figure out what happens here

Type Narrowing: Dropping Bits

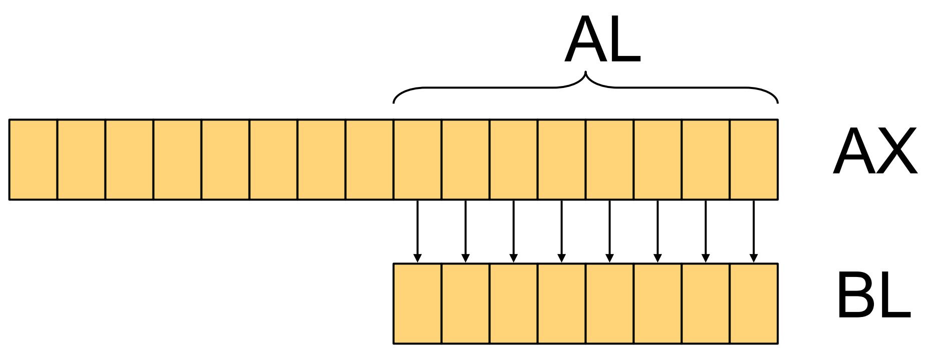
- Type narrowing: drop the most significant byte(s) and keep the least significant byte(s)
- Example: casting from a 4-byte int to a 2-byte int



- How would we do this in assembly?

Type Narrowing in Assembly

- We can use the the fact that we can access lower bits of some registers
- Example:
  - mov AX, [L] ; load 16 bits in AX
  - mov BL, AL ; take the lower 8 bits of AX and puts them in BL



- We have “cast” a 2-byte value into a 1-byte value
- If the 2-byte value is in a register like ESI or EDI, then we’d have to move the value into a register where we can access the lower bytes
- If the 2-byte value is in memory, then we could “just” read the 1-byte value into a 1-byte register (but watch out for Little Endianness!!)

Type Narrowing Correctness?

- When doing type narrowing **one loses bits**, and thus perhaps information
- Based on our signed / unsigned interpretation of the number, then we may get a result that is not equal numerically to the original number
- Let's consider the following 2-byte values, which we cast into 1-byte values:
  - $0051_{16} \Rightarrow 51_{16}$
  - $FFA2_{16} \Rightarrow A2_{16}$
  - $00B1_{16} \Rightarrow B1_{16}$
  - $FF7A_{16} \Rightarrow 7A_{16}$
- Which ones of the above make sense numerically?

Type Narrowing Correctness?

Unsigned		
2-byte	1-byte	
0051 (81_{10})	51 (81_{10})	✓
FFA2 (65442_{10})	A2 (162_{10})	✗
00B1 (177_{10})	B1 (177_{10})	✓
FF7A (65402_{10})	7A (122_{10})	✗

Signed		
2-byte	1-byte	
0051 (81_{10})	51 (81_{10})	✓
FFA2 (-95_{10})	A2 (-95_{10})	✓
00B1 (177_{10})	B1 (-79_{10})	✗
FF7A (-134_{10})	7A (122_{10})	✗

Type Narrowing Correctness?

Unsigned		
2-byte	1-byte	
0051 (81_{10})	51 (81_{10})	✓
FFA2 (65442_{10})	A2 (162_{10})	✗
00B1 (177_{10})	B1 (177_{10})	✓
FF7A (65402_{10})	7A (122_{10})	✗

Values are too large to be encoded with only 8 bits (we lost bits that were set to 1)

Signed		
2-byte	1-byte	
0051 (81_{10})	51 (81_{10})	✓
FFA2 (-95_{10})	A2 (-94_{10})	✓
00B1 (177_{10})	B1 (-79_{10})	✗
FF7A (-134_{10})	7A (122_{10})	✗

We lost mostly “useless” bits, but the remaining sign bit is wrong, so the results is wrongly positive or negative

Two "Rules" to Remember

- For unsigned numbers: size reduction leads to a numerically valid result if all removed bits are 0

0	0	0	0	0	0	0	0	X	X	X	X	X	X	X	X

X	X	X	X	X	X	X	X

- For signed numbers: size reduction leads to a numerically valid result if:
  - All removed bits are all 0's or if all removed bits are all 1's

    AND
  - The sign bit (the most significant bit) is unchanged

a	a	a	a	a	a	a	a	a	X	X	X	X	X	X	X

a = 0 or 1

a	X	X	X	X	X	X	X

Type Narrowing in High-Level PLs

```
int a = 65535;         // 4-byte (0x0000FFFF)
short b = a;
printf("%d\n",b);   // prints ????


int x = -50000;        // 4-byte (0xFFFF3CB0)
short y = x;
printf("%d\n",y);   // prints ????
```

- Any ideas?

Type Narrowing in High-Level PLs

```
int a = 65535;        // 4-byte (0x0000FFFF)
short b = a;
printf("%d\n",b);    // prints -1


int x = -50000;       // 4-byte (0xFFFF3CB0)
short y = x;
printf("%d\n",y);    // prints 15536
```

- Same outcome for the Java version (of course!)

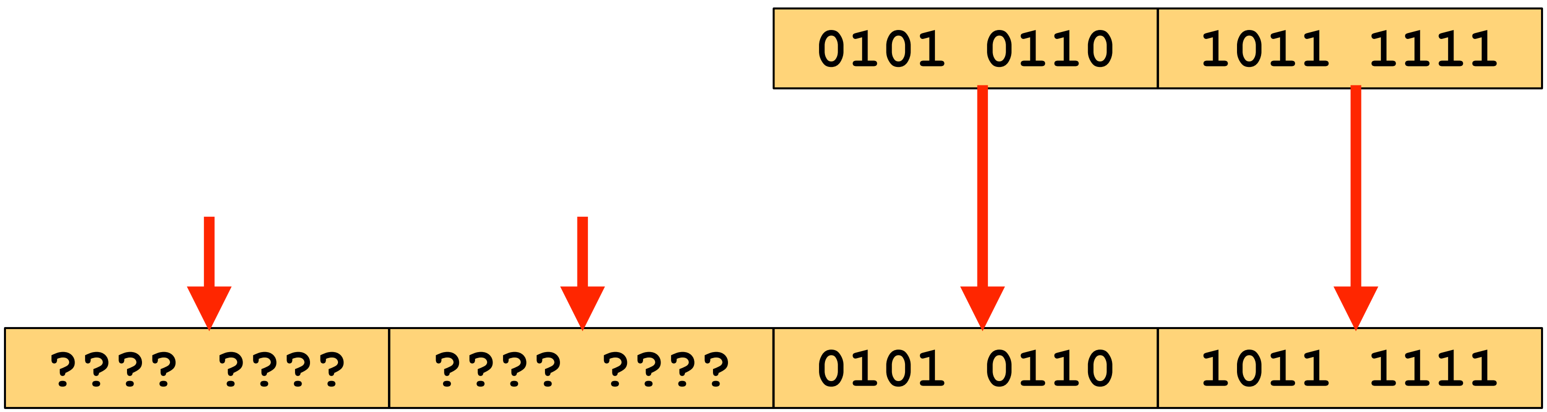
Type Widening: Size Increase

- Sometimes we need to increase the size of values using a cast
- This is called type widening

```
short a = -60;     // 2-byte, signed
int b = a;
printf("%d\n",b);  // prints ????


unsigned short x = 12;    // 2-byte, unsigned
unsigned int y = x;
printf("%d\n",y);  // prints ????
```

Type Widening: Adding Bits

- Type widening: add most significant bits
- Example: casting from a 2-byte int to a 4-byte int



- What should the new bits be?

Unsigned/Signed Type Widening

- **Unsigned** quantities: just add a bunch of 0's
- **Signed** quantities: perform sign extension
  - Add a bunch of replicas of the sign bit

- **Unsigned**

1101 0110	1011 1111

0000 0000	0000 0000	1101 0110	1011 1111

- **Signed**

1101 0110	1011 1111

1111 1111	1111 1111	1101 0110	1011 1111

0101 0110	1011 1111

0000 0000	0000 0000	0101 0110	1011 1111

Unsigned/Signed Type Widening

- **Unsigned** quantities: just add a bunch of 0's
- **Signed** quantities: perform sign extension
  - Add a bunch of replicas of the sign bit

- **Unsigned**

		D 6	B F
0 0	0 0	D 6	B F

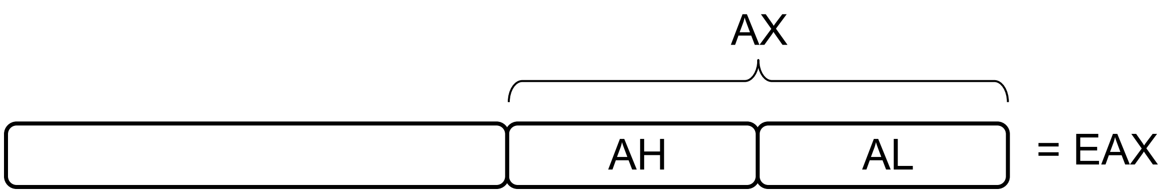
- **Signed**

		D 6	B F
F F	F F	D 6	B F

		7 6	B F
0 0	0 0	7 6	B F

Unsigned Type Widening in Assembly

- To increase a 1-byte value into a 2-byte value, one can play a trick:
  - Put the 1-byte value into AL, set AH to 0, AX now contains the 2-byte value
- But not for casting a 2-byte value into a 4-byte value: there is no way to access the 16 high bit of register eax separately!



- There is an instruction called **`movzx`** (Zero eXtend), which takes two operands:
  - Destination: 16- or 32-bit register
  - Source: 8- or 16-bit register, or 1 byte in memory, or 1 word in memory
  - The destination must be larger than the source!

Using movzx

- `movzx eax, ax ; zero extends ax into eax`
- `movzx eax, al ; zero extends al into eax`
- `movzx ax, al  ; zero extends al into ax`
- `movzx ebx, ax ; zero extends ax into ebx`
- `movzx ebx, [L]; leads to a “size not specified” error`

- `movzx ebx, byte [L] ; zero extends 1-byte value at address L into ebx`
- `movzx eax, word [L] ; zero extends 2-byte value at address L into eax`

Signed Type Widening in Assembly

- There is no (easy) way to use **`mov`** or **`movzx`** instructions to increase the size of signed numbers, because of the needed sign extension
  - Sometimes we want to add 0's (like **`movzx`**), but sometimes we want to add 1's (unlike **`movzx`**)

- For this reason, we have a new instruction: **`movsx`** (Sign eXtend)
  - Works just like **`movzx`**, but does sign extension

- Let's see an example..

Example

```
mov al, 0A7h ; as a programmer, I view this
             ; as an unsigned, 1-byte
             ; quantity (decimal 167)

mov cl, 0A7h ; as a programmer, I view this
             ; as a signed 1-byte
             ; quantity (decimal -89)

movzx eax, al ; extend to a 4-byte value
              ; (000000A7)
movsx ebx, cl ; extend to a 4-byte value
              ; (FFFFFFA7)
```

In-class Exercise

- Consider the following code

```
mov      al, 0B2h
movsx    eax, al
mov      bx, ax
movzx    ebx, bx
```

- What’s the final value of eax?
- What’s the final value of ebx?

In-class Exercise Solution

Instruction	EAX				EBX			
`mov al, 0B2h`	??	??	??	B2	??	??	??	??
`movsx eax, al`	FF	FF	FF	B2	??	??	??	??
`mov bx, ax`	FF	FF	FF	B2	??	??	FF	B2
`movzx ebx, bx`	FF	FF	FF	B2	00	00	FF	B2

Type Widening in High-Level PLs

```
short a = -60;      // 0xFFC4
int b = a;
printf("%d\n",b);   // prints ????


unsigned short x = 12; // 0x000C
unsigned int y = x;
printf(“%d\n",y);   // prints ????
```

- Any ideas?

Type Widening in High-Level PLs

The compiler will generate a **movsx** instruction

```
short a = -60;      // 0xFFC4
int b = a;
printf("%d\n",b);  // prints -60 (0xFFFFFFC4)


unsigned short x = 12; // 0x000C
unsigned int y = x;
printf("%d\n",y);  // prints 12 (0x0000000C)
```

The compiler will generate a **movzx** instruction

Another Example

```
unsigned char uchar = 0xFF;
signed char schar = 0xFF;
int a = uchar;
int b = schar;

printf("%d\n",a);  // prints ????
printf("%d\n",b);  // prints ????
```

- Any ideas?

Another Example

```
unsigned char uchar = 0xFF;
signed char schar = 0xFF;
int a = uchar;
int b = schar;

printf("%d\n",a);  // prints 255
printf("%d\n",b);  // prints -1
```

The compiler with generate a **movzx** instruction

The compiler with generate a **movsx** instruction

How `printf` works

- By declaring variables as “signed” or “unsigned” you define which of **`movsx`** or **`movzx`** will be used for type widening
  - This happens when you do an implicit or explicit cast
- It also happens with **`printf`**, **which does type widening when necessary!**
- Say you pass a n-byte argument x to **`printf`**, and that you print it using some “%X” format string
- What **`printf`** does is:
  - If “%X” is for a larger data size, then printf uses **`movzx`** or **`movsx`** to cast the value to a larger size **based on the value’s declared data type**
  - Then the number is printed based on the “%X” format string, **regardless** of how it was declared
- Good luck understanding this if you never studied assembly!
- Let’s see a few examples…

Understanding `printf`

```
unsigned char uc = 0xA0; // hex
printf("%d\n", uc); // prints 160
```

- The argument to **`printf`** is a 1-byte value
- The format string "%d" is for a 4-byte value
- So **`printf`** will increase the size to a 4-byte value
- Variable **`uc`** is declared as an unsigned value
- So the size increase will be done using **`movzx`**
- So the hex value will be 00 00 00 A0
- **`printf`** then interprets this value as a signed value (since the format string is "%d" and not "%u")
- So so, the above prints 160

Understanding `printf`

```
signed char sc = 0xA0; // hex
printf(“%u\n”, sc); // prints a large >0 number
```

- The argument to **`printf`** is a 1-byte value
- The format string “%u” is for a 4-byte value
- So **`printf`** will increase the size to a 4-byte value
- Variable **`uc`** is declared as a signed value
- So the size increase will be done using **`movsx`**
- So the hex value will be FF FF FF A0
- **`printf`** then interprets this value as an unsigned value (since the format string is “%u” and not “%d”)
- So the above prints some huge number (4294967200)

In-Class Exercise

```
unsigned short us = 259; // 0x0103
signed short ss = -45; // 0xFFD3

printf("%d %d\n", us, ss);  // prints ????
printf("%u %u\n", us, ss);  // prints ????
```

- What does the above print?

Solution

```
unsigned short us = 259; // 0x0103
signed short ss = -45; // 0xFFD3

printf("%d %d\n", us, ss);  // 259 -45
printf("%u %u\n", us, ss);  // 259 4294967251
```

A “kitchen sink” example

```
unsigned short ushort; // 2-byte quantity
signed char schar;     // 1-byte quantity
int integer;           // 4-byte quantity

schar = 0xAF;
integer = (int) schar;
integer++;
ushort = integer;

printf(“%d\n”, ushort);  // prints ????
```

- What does this code print?
  - Or what’s the hex value of the value it prints?
- Let’s do this together…

A “kitchen sink” example

```
unsigned short ushort;
signed char schar;
int integer;

schar = 0xAF;

integer = (int) schar;

integer++;

ushort = integer;

printf(“%d\n”, ushort);
```

schar	AF			
integer	FF	FF	FF	AF
integer	FF	FF	FF	B0
ushort	FF	B0		

Because **`printf`** doesn’t specify “h” ushort is size augmented to 4-bytes using **`movzx`** (because declared as unsigned): 00 00 FF B0
The number is then printed as a signed integer (“%d”): 65456

More Signed/Unsigned in C

- On page 32 of the textbook there is an interesting example about the use of the fgetc() function
  - fgetc reads a 1-byte character from a file but returns it as a 4-byte quantity!
- This is a good example of how understanding low-level details can be necessary to understand high-level constructs
- Let’s go through the example...

The Trouble with `fgetc`

- The **`fgetc`** function in the standard C I/O library takes as argument a file opened for reading, and returns a character, i.e., an ASCII code
- This function is often used to read in all characters of the file
- The prototype of the function is:

```
int fgetc(FILE *)
```

- One may have expected for **`fgetc`** to return a char rather than an int, since it's used to "get a character"
- But if the end of the file is reached, **`fgetc`** returns a special value called **`EOF`** (End Of File)
  - Typically defined to be -1 (**`#define EOF -1`**)
- So **`fgetc`** returns either
  - A character zero-extended into a 4-byte int (i.e., 000000xx), or
  - Integer -1 (i.e., FFFFFFFF)

The Trouble with `fgetc`

- Buggy code to compute the sum of ASCII codes in a text file:

```
char c;
while ((c = fgetc(file)) != EOF) {
        sum += c;
}
```

- In this code we have mistakenly declared c as a char
- C being C (and not Java), it thinks we know what we're doing and does a type narrowing of a 4-byte int into a 1-byte char when doing the assignment into **`c`**
- Let's say we just read in a character with ASCII code FF (decimal 255, "ÿ")
- **`fgetc`** returned 000000FF, but it was truncated into 1-byte integer **`c=0xFF`**
  - FF is -1 in decimal
- So we then compare 1-byte value FF to 4-byte value FFFFFFFF
  - C allows comparing signed integer values of different byte sizes, for convenience, by internally sign-extending the shorter value
    - **`int x=-1; char y=-1;  // (x == y) returns TRUE`**
  - So FF is sign-extended into FFFFFFFF
- Therefore, the above code will "miss" all characters after ASCII code FF and mistake them for an end of file
- Solution: declare **`c`** as an **`int`** (which may seem counter-intuitive)

Example: Type Widening Bug

- If you search around, you'll find bug reports about type widening pretty frequently
- For instance for crypt_blowfish: https://unspecified.wordpress.com/2011/08/08/integer-conversions-in-c/
  - "There's an implicit type widening of a signed char, that then can add a bunch of 1's when the intent was to always add a bunch of 0's"
  - "This can result in passwords being even easier to crack than expected. This is due to a char signedness bug in crypt_blowfish."
- There are many more recent examples out there
  - https://sourceforge.net/p/perfmon2/bugs/11/
  - "I was able to squelch the error by adding a cast everywhere: = (unsigned long) -1;"

Should you care?

- It all depends of what kind of work you do and what kind of software you deal with
  - Some codes will have stuff like that all over with signed/unsigned declarations and casts galore
  - Some codes will have none of that ever
- If all you do is JavaScript Web app development, you likely will rarely care
- If you do lower-level development, you may care every single day
  - Or rather, if you don't know all this, your life will be very difficult for understanding, debugging, etc.
- Overall, it's difficult to completely avoid it completely
  - In part due to binary data formats used all over the place

Important Takeaways

- **TLDNR:** A cast in C is not a conversion: It's a reinterpretation of the same bits, possibly with bits dropped or added
- Casting can be used for decreasing data size (type narrowing)
  - Just drop bits
  - For unsigned values, correct if dropping only zeros
  - For signed values, correct if dropping only all zeros or all ones, without changing the sign of the value
- Casting can be used for increasing data size (type widening)
  - Implemented via `movzx` or `movsx` instructions for unsigned or signed values
  - The compilers of high-level languages translate explicit/implement casts into `movzx` and `movsx` instructions
- Casting can be used to re-interpret bits (e.g., unsigned to signed)
- Casting is often needed but is a bit "sketchy" and shouldn't be overly used
- Printf is more complicated than you think :)

Conclusion

- Being aware of data sizes and of data size extension/reduction behaviors is important when doing lower-level development
- Unfortunately, almost every developer at some point is confronted with data size issues and having studied a bit of assembly is the only way to resolve mysteries
  - Important to know that a cast isn’t magical, and can do the "wrong” thing
- Let’s look at some of the practice problems…