Conversions between arithmetic types

- broadly speaking, the C compiler automatically converts arithmetic types in a similar way that the Java compiler converts types
  - in an expression such as `z = x + y;` where `x` and `y` have different types, the narrower type is converted to the wider type
- the presence of unsigned and signed types in C, as well as the lack of specificity in the sizes of the types, is a complicating factor

Integer conversion rank

- every integer type has a *conversion rank*

Rank from greatest to least (top to bottom)		
long long int	unsigned long long int	
long int	unsigned long int	
int	unsigned int	
short int	unsigned short int	
char	unsigned char	signed char
_Bool		

Integer promotion

- when performing arithmetic with a type whose rank is less than the rank of **`int`**, integer promotion occurs

Rank from greatest to least (top to bottom)		
`long long int`	`unsigned long long int`	
`long int`	`unsigned long int`	
`int`	`unsigned int`	
`short int`	`unsigned short int`	
`char`	`unsigned char`	`signed char`
`_Bool`		

types in red are called *small types*

Integer promotion

- integer promotion is the process of automatically converting a small type to **`int`** or **`unsigned int`** when performing an arithmetic operation
- allows operations to be performed using the optimal integer size for the architecture
- helps to prevent intermediate overflow errors

In the example below, the intermediate sum overflows the range of signed char, but promotion allows the sum and final result to be computed correctly.

```
#include <limits.h>
#include <stdio.h>

int main(void) {
    signed char max = SCHAR_MAX;
    signed char c1 = 100;
    signed char c2 = 50;
    signed char n = 2;
    signed char avg = (c1 + c2) / 2;    // no overflow!

    printf("max char: %d\n", max);
    printf("avg     : %d\n", avg);
    return 0;
}
```

Mixed operand types

- when mixed types are used in an arithmetic expression, one or both operands are converted according to the following four cases which are checked in order:
  1. one of the operands has type long double
  2. one of the operands has type double
  3. one of the operands has type float
  4. both operands are some integer type

Cases 1, 2, and 3

1. if one of **`x`** and **`y`** has type **`long double`** then the other operand is converted to **`long double`**
2. otherwise, if one of **`x`** and **`y`** has type **`double`** then the other operand is converted to **`double`**
3. otherwise, if one of **`x`** and **`y`** has type **`float`** then the other operand is converted to **`float`**

```c
#include <float.h>      // floating point limits
#include <limits.h>     // integer limits
#include <stdio.h>
int main(void) {
    long double x = DBL_MAX;
    double y = DBL_MAX;
    long double sum = x + y;
    printf("DBL_MAX : %le\n", DBL_MAX);       // le : double to exponential notation
    printf("sum     : %Le\n", sum);           // Le : long double to exponential notation

    y = FLT_MAX;
    float z = FLT_MAX;
    sum = y + z;
    printf("FLT_MAX : %e\n", FLT_MAX);        // e : float to exponential notation
    printf("sum     : %Le\n", sum);

    z = SHRT_MAX;
    short s = SHRT_MAX;
    sum = z + s;
    printf("SHRT_MAX : %d\n", SHRT_MAX);
    printf("sum      : %Lf\n", sum);

    return 0;
}
```

Case 4

- if both operands are of integer type, then integer promotion is applied if required
  - see example four slides previous

Mixed integer operands

- after promotion occurs, conversions are automatically applied according to five separate cases which are checked in order
  1. **x** and **y** have the same type
  2. **x** and **y** are both signed types, or both unsigned types
  3. the unsigned operand has rank greater than or equal to the signed operand
  4. the signed operand type can represent all the values of the unsigned operand
  5. the catchall case

Case 1

- if **x** and **y** have the same type, then no further conversions are applied

Case 2

- if **x** and **y** are both signed types, or both unsigned types, then the operand with the lower rank is converted to the type of the other operand

Case 3

- if the unsigned operand has rank greater than or equal to the signed operand, then the signed operand is converted to the type of the unsigned operand
  - the results are often very surprising if the signed operand is negative
    - why?

Case 4

- if the signed operand type can represent all the values of the unsigned operand, then the unsigned operand is converted to the type of the signed operand
  - occurs when the signed type is wider than the unsigned type

Case 5

- occurs when:
  - two different types are represented with the same number of bits, and
  - the signed operand has greater rank than the unsigned operand
- both operands are converted to the unsigned type of the operand having the signed type
  - the results are often very surprising if the signed operand is negative

```c
#include <limits.h>
#include <stdio.h>
int main(void) {
    long int x = INT_MAX;
    int y = INT_MAX;
    unsigned long int sum = x + y;              // case 2  long int + int
    printf("INT_MAX : %d\n", INT_MAX);
    printf("sum     : %lu\n", sum);

    unsigned int z = INT_MAX;
    sum = 1 + z;                                // case 3  unsigned int + int
    printf("INT_MAX : %d\n", INT_MAX);
    printf("sum     : %lu\n", sum);

    y = USHRT_MAX;
    unsigned short s = USHRT_MAX;
    sum = s + y;                                // case 4  unsigned short + int
    printf("USHRT_MAX : %u\n", USHRT_MAX);
    printf("sum       : %lu\n", sum);

    return 0;
}
```

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

// unsigned_conversion.c

int main(void) {
    unsigned int ui = UINT_MAX;
    signed char c = -1;
    if (c == ui) {
        printf("%d equals %u\n", c, ui);
    }

    return 0;
}
```

Safe conversions

- a signed integer of a smaller type can be safely converted to a signed integer of a larger type
- an unsigned integer of a smaller type can be safely converted to a unsigned integer of a larger type
- a **`float`** can be safely converted to **`double`** or **`long double`**
- a **`double`** can be safely converted to **`long double`**

Floating-point to integer conversion

- except for `_Bool`:
  - a floating-point value is converted to an integer type by discarding the fractional part of the floating-point value
  - if the whole number part of the floating-point value cannot be represented by the integer type, then the behavior is undefined

```c
#include <float.h>
#include <limits.h>
#include <stdio.h>

// float2int.c

int main(void) {
    float x = 1.5;
    int y = x;
    printf("y = %d\n", y);

    x = FLT_MAX;
    y = x;
    printf("y = %d\n", y);

    return 0;
}
```

Integer to floating-point conversion

- if the value of an integer fits in the range of a floating-point type, then the integer value is converted to the closest possible floating-point value
  - **`float`** often cannot represent every **`int`** value exactly
  - **`double`** often cannot represent every **`long`** value exactly
- otherwise, the behavior is undefined

```c
#include <float.h>
#include <limits.h>
#include <stdio.h>

// int2float.c

int main(void) {
    int x = 101;
    float y = x;
    printf("x = %d, y = %f\n", x, y);

    x = INT_MAX - 100;
    y = x;
    printf("x = %d, y = %f\n", x, y);

    return 0;
}
```

Floating-point to floating-point conversion

- if the value of a floating-point value fits in the range of the target floating-point type, then the value is converted to the closest possible floating-point value of the target type
- otherwise, the behavior is undefined

```c
#include <float.h>
#include <limits.h>
#include <stdio.h>

// double2float.c

int main(void) {
    double x = 100.5;
    float y = x;
    printf("x = %f, y = %f\n", x, y);

    x = FLT_MAX * 2.0;
    y = x;
    printf("x = %f, y = %f\n", x, y);

    return 0;
}
```

Type qualifiers

Type qualifiers

- types can be qualified using one of four different keywords:
  - **`const`**
  - **`volatile`**
  - **`restrict`**
  - **`_Atomic`** (C11)
- only **`const`** is of interest to us for the purposes of CISC220

`const`

- objects declared with the **`const`** qualifier are not modifiable
  - cannot assign a value to **`const`** variable, but it can be given an initial value
- the compiler is allowed to place a **`const`** qualified object in read-only memory

The following program does not compile because the programmer tries to assign a value to **`const`** qualified variable.

```c
// const.c
int main(void) {
    const int i = 1;
    i = 2;

    return 0;
}
```

`const`

- you can create a pointer to a **`const`** qualified object
- a modern compiler will warn you that the **`const`** qualifier is lost if you do
  - the result is undefined behavior if you modify the object via the pointer

The following program compiles with a warning, but it does compile successfully. The runtime behavior is undefined.

```c
#include <stdio.h>
// const2.c
int main(void) {
    const int i = 1;
    int *p = &i;
    *p = 2;
    printf("i = %d\n", i);

    return 0;
}
```

`const`

- to create a pointer to a **`const`** qualified object you should qualify the pointer declaration with **`const`**
- the compiler will then prevent you from trying to change the object pointed to
- the declaration

```
const int *p;
```

declares a pointer to a constant **`int`** object

The following program does not compile because the programmer tries to assign a value to **`const`** qualified variable **`i`** via a pointer to **`const`**.

```c
#include <stdio.h>
// const3.c
int main(void) {
    const int i = 1;
    const int *p;
    p = &i;
    *p = 2;
    printf("i = %d\n", i);

    return 0;
}
```

`const`

- if you want the pointer to be constant (i.e., you want the pointer to always point to the same object) then you should use the following syntax:

  ```
  int * const p = &some_int;
  ```

  which declares a constant pointer to an **`int`** object

The following program does not compile because the programmer tries to assign a value to **`const`** qualified variable **`p`**.

```
#include <stdio.h>
// const4.c
int main(void) {
    const int i = 1;
    int * const p = &i;         // compiler warning
    p = &i;                     // compiler error

    printf("i = %d\n", i);

    return 0;
}
```

`const`

- if you want the pointer to be constant and the object pointed to be constant then write:

  ```
  const int * const p = &some_int;
  ```

  which declares a constant pointer to a constant **`int`** object

Expressions and operators

Operators

- most of the operators that use in Java have the same meaning in C
- a notable exception is the remainder operator **`%`**
  - in C, **`%`** works only with integer types
  - in C, the sign of **`x % y`** is always equal to the sign of **`x;`** the following are all true
    - ```
       10 %  3 ==  1
       10 % -3 ==  1
      -10 %  3 == -1
      -10 % -3 == -1
      ```
      ```

Operators

- another notable exception is that multiple comparisons such as

```
x < y < z
```

  are allowed, but it does not mean what you probably think it means
- the above expression is interpreted as

```
(x < y) < z
```

  i.e., **`z`** is compared to **`0`** or **`1`**