Typecasting and Formatting

Class 8

Skipped Content

- we will not explicitly cover:
  - the material from page 122 to the middle of page 126: cin.get, cin.ignore, string functions
  - the material from the bottom of page 126 to the bottom of 128: additional math library functions (but look at the table on page 127 to see that the functions exist)
  - sections 3.10 and 3.11 — please read over them, but we won't cover them in a lecture and they will not specifically be on the test

Numerical Types

- remember there are three fundamentally different families of numerical data types
- they have very different purposes

Family	Purpose
unsigned integers	counted quantities
signed integers	whole numbers that might need to be negative
floating point	measured or calculated quantities that might have fractional parts

Keep Data Types Separate

- the arithmetic operators are defined for identical data types
  - `unsigned = unsigned + unsigned;`
  - `double = double + double;`
- to the greatest extent possible, you should avoid mixing data types in expressions

- however, sometimes you must mix data types in a single expression
- the compiler has a set of rules to try to convert one into the other
- the purpose of the rules is to avoid information loss

Type Ranking

- C++ ranks types by the largest value each can hold
  1. long double
  2. double
  3. float
  4. unsigned long
  5. long
  6. unsigned
  7. int
  8. unsigned short
  9. short

Terminology

coercion:	convert a value of one type to a different type (floating $\leftrightarrow$ integral or signed $\leftrightarrow$ unsigned)
promotion:	convert a value to a higher-ranked type
demotion:	convert a value to a lower-ranked type

Mixing Sizes

- remember some of the integer sizes

Name	# Bytes (ice)	Range of Values
short	2 bytes	$-32,768\ldots32,767$
unsigned short	2 bytes	$0\ldots65,535$
int	4 bytes	$-2,147,483,648\ldots2,147,483,647$
unsigned int	4 bytes	$0\ldots4,294,967,296$

- a signed short's value can always fit into an int location
- an unsigned short's value can always fit into an int location
- a signed short's value might not fit into an unsigned int location
- an int value might not fit into an unsigned int location
- an unsigned int value might not fit into an int location

Mixing Sizes

- the compiler will not allow an attempt to convert to a type that might not be able to hold the value

```
int foo = 10;
unsigned bar = foo;
warning: implicit conversion changes
   signedness: 'int' to 'unsigned int'

int foo = 10;
short bar = foo;
warning: implicit conversion loses
   integer precision: 'int' to 'short'

float foo = 10.0;
int bar = foo;
warning: implicit conversion turns
   floating-point number into integer: 'float' to 'int'
```

Mixed Types

- there are several automatic conversions that it is ok to use
- the compiler does the conversions for you
- this differs somewhat from what your textbook says
- the clang-llvm compiler is much more strict than older, classic compilers

- the following pairs of mixed types are "safe"
- but you still need a good reason to mix them

Types	**Result Type**
two signed integer types	the larger type
two unsigned integer types	the larger type
an integer type and a floating type	the floating type

Mixed Types

- thus the following are legal:

```
short value1 = 10;
int value2 = value1; // signed integer -> bigger location
double value3 = value2; // integer -> floating point type
```

- even though the third line is very questionable
- however, clang does not allow the following, even though it "seems" ok:

```
float value1 = 123.45F;
double value2 = value1;
```

```
warning:  implicit conversion increases
 floating-point precision:  'float' to 'double'
```

Concise

- there is a fine line between being concise and being sloppy
- being concise involves
  - keep it short
  - don't use more words if fewer words will suffice
  - don't use a longer expression if a shorter one gets the same results
- however, sometimes being short is not concise, it's sloppy:
  `double weight_of_material = 0;`
- weight_of_material is declared as a double because it will involve a measured quantity
- a double has a whole part and a fractional part
- the correct initialization is:
  `double weight_of_material = 0.0;`
- this is a signal that you the programmer are consciously choosing the correct data type

Type Casting

- sometimes you need to mix types that are "unsafe"
- sometimes you need to explicitly convert types
  1. you need to convert an integer into a floating point to perform floating point division
  2. the compiler would not normally allow an automatic conversion, but you the programmer know it is safe

Typecasting 1

- calculate a floating point average value, given two integers
  `double average = tantrum_sum / NUMBER_OF_VALUES;`
- no errors or warnings
- integer division (truncates)
- result has no fractional part, so it's the "wrong" answer
- solution: typecast

```
double average =
 static_cast<double>(tantrum_sum) / NUMBER_OF_VALUES;
```

Typecasting 2

- need to put a signed integer into an unsigned location
- imagine a series of calculations that results in a value that cannot be negative
- perhaps involving squaring values or absolute values
- you logically know the result is non-negative
- but the compiler can't read your mind
- you can force the conversion:

```
unsigned bar = static_cast<unsigned>(foo);
```

Overflow and Underflow

- what does the following program output?

```
unsigned short foo = 0;
foo -= 1;
cout << foo << endl;

short bar = 32767;
bar += 1;
cout << bar << endl;
```

Overflow and Underflow

- what does the following program output?

```
unsigned short foo = 0;
foo -= 1;
cout << foo << endl;

short bar = 32767;
bar += 1;
cout << bar << endl;
```

- overflow is when a value is generated that is too large to fit into its type
- underflow is when a value is generated that is too small to fit into its type

Overflow and Underflow

- on ice with the clang-llvm compiler
- integer overflow and underflow wrap around to the other side
- no error

- floating point overflow results in “inf” or “$-$inf”
- you can output this value
- any subsequent calculations with this value remain inf

- floating point underflow results in 0
- you can subsequently calculate with this value

Formatting Integer Output

- the default way cout displays an integer value is to display just the base-10 digits using as many columns as there are digits in the value
- if it is a negative value, there is a unary minus in the column before the first digit
- there are three common ways this is sometimes modified
  - the number of columns, the width, taken up by the value can be increased (but not decreased)
  - the value can be left justified within the width instead of the default right justification
  - the padding character printed in the non-digit spaces can be changed from the default space character

program integer_format.cpp

iomanip Library

- the library has many functions
- we will only use a few
- most of the functions are sticky
- they persist for all output until changed
- only setw needs to be repeated, each time

Default Floating Point Output

- use fixed (non-scientific) notation for values between approximately ± 0.00001 and ± 999999.9 (varies among different computers)
- scientific notation for values smaller or bigger than this
- do not show a decimal point or fractional part if the value has no fractional part within the default width
- right-justify the output within the width, padding with spaces if necessary

Floating Point Manipulators

Manipulator	Description
setw	minimum number of columns used
setprecision(n)	value is rounded to at most n significant digits (perhaps switching to scientific notation to do so)
fixed	force non-scientific notation
showpoint	force showing decimal point and at least one fractional digit
left	left-justify the output within width columns

- all except setw are sticky

setprecision

- setprecision is a complicated manipulator
- by itself, it sets the maximum absolute number of significant digits
- when used with fixed, it changes and displays that number of digits after the decimal point
- setprecison plus fixed implies showpoint
- for many situations, setprecision plus fixed is the correct combination for showing “normal” floating point values

cin Input with Embedded Spaces

- we have seen that cin extraction always stops at whitespace
- sometimes we need to read a string from the keyboard that has embedded spaces
- for example, a person's full name
- we can do this with the getline function: `getline(cin, variable);`
- getline reads from wherever the keyboard buffer cursor is to the next newline
- getline does not skip whitespace or any other characters

see program using_getline.cpp

Pseudorandom Numbers

- in section 3.9 Gaddis introduces C++'s pseudorandom number generator
- it is a simple system, good enough for games, but not nearly strong enough for cryptography or security

Pseudorandom Numbers

- in section 3.9 Gaddis introduces C++'s pseudorandom number generator
- it is a simple system, good enough for games, but not nearly strong enough for cryptography or security

- the main feature of the system is the function rand() which returns a pseudorandom number between 0 and a large integer
- each subsequent call to rand returns another value in that range

Pseudorandom Numbers

- in section 3.9 Gaddis introduces C++'s pseudorandom number generator
- it is a simple system, good enough for games, but not nearly strong enough for cryptography or security

- the main feature of the system is the function rand() which returns a pseudorandom number between 0 and a large integer
- each subsequent call to rand returns another value in that range

- each time the program runs, the same sequence of values is returned by rand!

Seeding the RNG

- while developing and testing, you may want each run of the program to produce the same sequence of values
- but for production use, you usually want a non-predictable sequence of values
- to accomplish this, you must seed the random number generator with a unique value
- this is done with the srand() function

- the normal way to generate a unique seed is to call the time() function, which gives the number of seconds between the specified date-time and midnight, 1 January 1970
- if time() is called with no parameter, it gives the number of seconds between the current time and midnight, 1 January 1970
- thus the normal way to use the RNG system is to call srand once, and then repeatedly call rand for each desired random number

Gaddis' Code

- the random system is a bit of a hodgepodge of ineptly mixed types
- Gaddis presents code in program 3-25 to use random, but it does not work as written because of type mixing problems
- the problem is with generating the seed where Gaddis has:
  ```
  unsigned seed = time(0);
  ```
- this generates two warnings due to unsafe type mixing
- instead, the code needs to be:
  ```
  unsigned seed =
   static_cast<unsigned>time(nullptr);
  ```

see program 3-25.cpp, corrected to fix Gaddis's type mixing problems, for a complete example

Using rand()

- rand() returns a signed int
- even though it only returns non-negative values
- sigh

- this is one of the inconsistencies of C++

Using rand()

- rand returns a value between 0 and a large integer value inclusive
- to instead pick a value between, say, 1 and 10 inclusive
- must define several constants as explained on page 130
- `MIN_VALUE` is the smallest value in the range (here, 1)
- `MAX_VALUE` is the largest value in the range (here, 10)
- the expression is:

  ```
  rand() % (MAX_VALUE - MIN_VALUE + 1) + MIN_VALUE;
  ```

- remember, this produces an int, even though the value is guaranteed to be non-negative
- to use it properly, you must typecast the resulting value to an unsigned