Types

James Daly
CSE 232
Michigan State University
Hello World with prompt

- Example 1.2
Chaining cout expressions

- Remember `<<` is a *binary* operator that returns the stream
  `cout << "Hello " << name << "!" << endl;`

  - *Do* `cout << "Hello "`
    - Print string, expression return cout stream
  - *The do next pair* `cout << name`
    - name is a string variable, print the string
  - *Then do* `cout << "!"
    - Single quotes, a single character
  - *Finally* `cout << endl`
Overloaded $\ll$ operator

- These three calls are to *three different functions* because of types
  - Print a constant string
  - Print a string
  - Print a character
endl

- The `endl` indicates you want the output to end the line and have the next output begin at the front of the next line.
- Does other things too (a flush) which we’ll discuss later.
3 ways to deal with std

- Three ways, one is **disallowed**
  - Merge all of `std` with the global namespace, using `namespace std;`
  - Indicate every time for each value the namespace it comes from
  - Declare up front only those particular elements you want to merge
Merging

- `using namespace std;`
- This essentially merges all the declarations in std into the global namespace.
  - No `std::` required anywhere
  - **Points off your project if you do this!**
Full merge is bad

- This is the easy way, but it is fraught with problems:
  - What just got merged (you don’t know)?
  - When you indicated a variable, what namespace did it come from?
  - Affects everyone who includes your file
Mark every variable with std

- If you mark each one, you can differentiate what namespace it belongs to
  `std::cout << "Hi mom" << std::endl;`
  - Allows for the same names from different namespaces
  - The most general way to go
  - Can get to be a pain
Merge only what you need

- You can get away with this
  ```
  #include<iostream>
  using std::cout;
  using std::endl;
  cout << "Hi Mom" << endl;
  ```
Hello world with name and age

- Example 1.3
Declaration

- Before you use a variable, you must declare it
  - At least say what type it will hold
    - Cannot change that variable’s type for the duration of the program
  - Could include an initial value
    - If you don’t, you get what the compiler decides is the default

- Different than Python
Extraction Operator

- For `cin` (input stream) we have the extraction operator (>>)
  - Pulls a **typed value** from the console input up to
    - white space
    - end of line
    - error
Typed value and cin

- When you run the extraction operator, cin is **overloaded** to deal with the type of variable the value is going into:
  - If it is an int, only read digits
  - If it is a float, read digits, ‘.’, ‘E’
  - If it is a string, reads anything
- If it hits a problem (read a float into an int) it reads what it can and then errors out
Other things in this version

- We included the string header. We could do STL string operations, but we just declared a string.
- Return value commented out (not required)
Things to note

- `cout` expression doesn’t have an `endl`
  - We can `cin` from the same line
- We have two declares
  - Integer age and string name
  - Didn’t give inits, takes defaults
    - 0 for int, "" for string
    - Questionable for int, compiler dependent
- Two different ops for `>>` (type dependent)
Spacing

- Shouldn’t use more than 80 columns on a line for readability
- Below is acceptable (note indentation)

```cpp
cout << "Hello " << name << ", " << 65 - age << " years to retirement" << endl;
```
Basic Types

- Example 2.1
Lots of types and modifiers

- We have to get the types right in C++
  - Compiled language needs to select the correct overloaded op at compile time
  - Provide aids to the programmer to control how information is moved about
Compiler is a program

- Two things
  - A compiler is another program. It translated code to something else (usually an assembly language)
    - It can make mistakes or have quirks
  - When you get down to blaming the compiler for your program’s errors you should probably call it a day
    - Likely it is you, not the compiler
Details of type can depend

- The C++ standard does not fully detail the required size of a type
  - It sets minimums or maximums
  - The compiler programmers are free to exceed those if they choose
  - You should run code on your compiler to see
<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>1 byte</td>
<td>Boolean (0, empty/false, everything_elses / true)</td>
</tr>
<tr>
<td>char</td>
<td>1 byte</td>
<td>Hold a character</td>
</tr>
<tr>
<td>short (short int)</td>
<td>2 bytes</td>
<td>±32,768</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes ((2^{32}))</td>
<td>Basic integer, (~\pm2\times10^9)</td>
</tr>
<tr>
<td>long (long int, long long)</td>
<td>8 bytes ((2^{64}))</td>
<td>64 bit integers, (~\pm9\times10^{18})</td>
</tr>
<tr>
<td>float</td>
<td>4 bytes</td>
<td>24 bits in significand</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
<td>53 bits in significand</td>
</tr>
<tr>
<td>long double</td>
<td>16 bytes</td>
<td>64 bits in significand</td>
</tr>
</tbody>
</table>
C++11

- To take advantage of the C++11 standard, you have to tell the compiler you are using code that is pursuant to that standard
  - Visual Studio: set the profile
  - In CLI, `g++ -std=c++11`
  - Example 2.1 requires it
Example 2.1

- Run it whenever you go to a new compiler to make sure you know what the basic types are!
Suggestions

- When in doubt:
  - Use `long` for an integer (if there is any chance of exceeding an `int`)
  - Use `double` for a float
- Especially true for doubles as floating-point numbers introduce all kinds of round-off errors
  - The more precision the better
Initialize Variables

- Example 2.2
Initialize Variables

- C++, because of its legacy support and feature creep has many ways to do things
- One of them is initialization of a variable
  - Some subtleties here
  - Let’s look at basics
Variants

- No init (compiler dependent)
- Assign init
- Parenthesis init(11)
- Curly init {11}

- There are some subtleties here that are worth noting (lots more later)
Three types

- These are initializations because they are *declaring* a variable

- Direction initialization (both equivalent)
  - `long my_long(my_int);`
  - `in my_int = 123;`

- Initializer list (depends on type)
  - `long another_long{1}`
C++ and efficiency

- Unlike in Python, in C++ we worry about efficiency
  - One of the main reasons to use C++
  - Can cause complications (but that is kind of the point)
- For efficiency’s sake, we want to avoid copies (because they are expensive)
What does = mean?

- Remember the context problem for C++
- The = (equal) sign means different things in different contexts

```cpp
int my_int = 23; // initialization
my_int = 123; // different op, assign
```
Expressions

- “Never express yourself more clearly than you are able to think.”

-Niels Bohr
Numeric Ops

- Example 2.3
Math operators

- Integers (all return integers)
  - Addition and subtraction: +, -
  - Multiplication: *
  - Division
    - / of two integers (returns an integer)
    - remainder: %

- Floating Point (all return floats)
  - Add, subtract, multiply, divide: +, -, *, /
Octal and Hex

- Pay attention to this

```cpp
int temp_int
temp_int = 010; // leading 0, octal
cout << temp_int; // prints 8
temp_int = 0x10; // 0x means hex
cout << temp_int; // prints 16
```
Type Conversion

- Converts one type to another
  - e.g. convert an integer to a floating point
  - Often called a *cast*
- There are a number of cast operators
- Right now we’ll talk about `static_cast`
  - Requires the “cast to” type in `< >`
  - `static_cast<int>(1.789) -> 1`
  - No rounding!
Automatic Cast

- When does C++ do an auto cast:
  - The binary operator you requested does not exist (the combination of types doesn’t exist)
  - There is a conversion operator of one of those types that works for an op
    - C++ tries to apply conversions that maintain information
  - In mixed math, int / long are auto cast to float / double
Integer Math

```cpp
int int2 = 2, int3 = 3;
double float3 = 3;
cout << int2 / int3; // ??
cout << int3 / int2; // ??
cout << int2 / float3; // ??
cout << int2 % int3; // ??
cout << int3 % int2; // ??
```
If no precedence, left to right in pairs

- $1 + 2 + 3 + 4$
  - $(1 + 2) + 3 + 4$
    - Addition returns a result, 3
  - $(3 + 3) + 4$
    - Addition returns a result, 6
- $6 + 4$
  - Returns 10
Assignment Ops

- Example 2.4
Assignment Expressions

- Format: \( lvalue = rvalue \)
- \( rvalue \) (rhs of \( = \)) represents a value
- \( lvalue \) (lhs of \( = \)) represents a memory location
- We are **copying** the value to the location
- Return value is the \( rvalue \)
Assignment Expression

- Follow precedence rules
- Example $x = 2 + 3 \times 5$
  - Evaluate the expression $(2 + (3 \times 5))$: 17
  - Change the value of $x$ to be 17
  - Return the value 17
- Example $(y$ has the value 2$): y = y + 3$
  - Evaluate expression $(y + 3)$: 5
  - Change the value of $y$ to be 5
  - Return the value 5
Chaining

- = is right associative
- Example: $x = y = 5$
- Behavior
  - Right associative $x = (y = 5)$
  - Expression $y = 5$ returns value 5
  - $x = 5$
A function / operator can do **two things**

- Perform some operation (write to output, change a variable’s value)
  - This is the **side-effect**
- **Return value** after the operation
  - Return can be assigned, etc.
Seen this in << operator

- `cout << whatever`
  - Side-effect, dump `whatever` to the `cout` stream
  - Return the stream (in this case `cout`)

- Allows for chaining

- `cout << 1 << 2`, pairs left to right
  - `cout << 1` -> returns `cout`
  - `cout << 2`
Shortcut: Increment

- Order (pre or post) matters. Side-effect the same, return value different
- Example: \( x = \text{++}y \);  
  - Pre-increment, return \textit{changed value}  
    - \( y = y + 1; \)  
    - \( x = y; \)
- Example: \( x = y++; \)  
  - Post-increment, return original value  
    - \( x = y; \)  
    - \( y = y + 1; \)
Other shortcuts

- **Decrement:** `--`
  - Example: `y = x--`

- **Compound assignment:**
  - `y += x` equivalent to `y = y + x`

- **Others**
  - `-=, *=, /=, %=`
Boolean Ops

- Example 2.6
Boolean Expressions

- **Value**: True or false
  - **Remnant of C**:
    - Integer value of 0 is equivalent to false
    - Nonzero integer value is equivalent to true
    - Both `true` and `false` are C++ terms
    - `true == 1`, `false == 0`

- **Example expression**: `age < 40`
  - **Format**: `expression op expression`
  - **Result**: 0, 1
Logical Operators

- Logical Operators
  - And: `&&`
  - Or: `||` (two vertical bar chars)
  - Not: `!`

- `(0 <= my_int) && (my_int <= 3)`
- `(0 <= my_int) || (my_int <= 3)`
- `!my_int`
## Truth Tables

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>( \neg p )</th>
<th>( p \land q )</th>
<th>( p \lor q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>
Alternative logical ops

- Turns out C++ does support `and`, `or`, and `not` as in Python
  - Your book doesn’t mention it
  - You are probably not in the C++ club if you do that
Relational Operators

- Example 2.7
Relational Operators

- Less than: `<`
- Greater than: `>`
- Equal to: `==` (not the same as `=`)
- Not equal to: `!=`
- Less than or equal to: `<=`
- Greater than or equal to: `>=`
Examples

- If the value of integer `my_int` is 5, the value of the expression `my_int < 7` is `true` (1)
- If the value of char `my_char` is ‘A’, then the value of the expression `my_char == ‘Q’` is `false` (0)
Pitfall

- Be careful of floating point equality comparison, especially with zero
  - e.g. `my_double == 0`
  - Float arithmetic is approximate
  - Use `!=` if you can
  - If not, use a tolerance
    - Value +/- the tolerance
Compound Expressions

- **Want:** $0 \leq my\_int \leq 3$ *(not like Python!)*
  - Consider $my\_int$ with value of 5
  - Left-associative: $(0 \leq my\_int) \leq 3$
  - $(0 \leq my\_int)$ is true, which has value 1
  - Therefore: $1 \leq 3$
  - Value of expression is true!
- **Solution:** $(0 \leq my\_int) \land (my\_int \leq 3)$
Three Things

- Assignments return a value!
- For each type
  - false: 0 / empty value
  - true: everything else
- Short circuiting
  - When it is “obvious” what a logical result will be, that result is returned and the compiler ignores the rest of the logical expression
short circuits on true

```cpp
int first = 0, second = 0
(first = 100) || (second = 200);
cout << "First:"<<first<<", Second:"<<second<<endl;
```

- What is the output?
&& short circuits on false

```cpp
int first = 0, second = 0
(first = 0) && (second = 200);
cout << "First:" << first << "", Second:" << second << endl;
```

- What is the output?
Intro to cout formatting

- Example 2.8
- See also
  - Table 17.17 on page 757
  - Table 17.18 on page 760
iostream manipulators

- Besides sending output (via `<<`) to `cout` or input (via `>>`) to `cin`, you can also set state in the stream
  - You set the stream to have a particular characteristic
  - State persists in the stream until you reset it
    - Mostly
iostream, for output

- **fixed**: fixed points for floats
- **scientific**: use scientific notation
- **setprecision**(prec): set the decimal points (with rounding) for floats (#include<iomanip>)
- **boolalpha/noboolalpha**: Show true or false for Booleans (0 or 1 otherwise)
More iostream, for output

- **left, right**: Flush output to the left or right (left or right justified)
- **showpoint, noshowpoint**: Always use a decimal point on output vs only have a decimal point when there is a fractional part
<iomanip>, for output

- **setw**(space_cnt)
  - Min width the output occupies
  - Does **not** set state, must be set for every field output
  - Wider if output is wider

- **setfill**(char)
  - In a wider field, fill with char
  - Space is default
Example 2.9
iostream for input

- `noskipws` or `skipws`
  - Do you count whitespace as a char
- `cin.eof()`
  - True if end-of-file character encountered
  - Different for each os
  - Ctrl-Z for Windows
  - Ctrl-D for Unix
cin is complicated

- When `cin` tried to read something into a type and cannot (or if it reads `EOF`), it goes into a fail state
  - Need to clear that fail state to keep going
  - `cin.clear()` does that
- This is complicated, we will discuss it more later
Clearing the cin buffer

- Once you clear the error you need to clean out the buffer so that you won’t get the same error
  `cin.ignore(char_num, delim)`
  - `cin.ignore(80, '\n')`, clear up to 80 chars or until '\n' is encountered
Assignment and If

- We haven’t seen if statements yet, but here is one anyway
  ```java
  int x = 5;
  if (x = 1)
    dosomething;
  ```
  That compiles fine, is always true, and probably not what you wanted
  (==)