Data Types and Classes

PIC 10A, UCLA

Data Types and Classes

Part of computing is recognizing what type of variable to use in a given situation. Here, we will look into the different types of variables we can create, including class objects, which can have extra functionalities that simple numerical types may lack.

Binary

We begin our study of data by looking into **binary**, a base 2 number system. This is important to understand because ultimately the only information that a computer can store is a series of on/off values, values of 1 and 0.

A **bit** is a single 0 or 1, off or on value.

An **octet** is a collection of 8 bits.

On many, but not all computer architectures, a **byte** is 8 bits. But this is not always true!!!

Binary

When we write a number in the usual **decimal** (base 10) system, we use 10 digits: 0 through 9. In writing the number **1403** , we mean the value

$1 \times 10^3 + 4 \times 10^2 + 0 \times 10^1 + 3 \times 10^0 =$
$1 \times 1000 + 4 \times 100 + 0 \times 10 + 3 \times 1.$

For notation, we may write $\mathbf{1403}_{10}$ to be explicit that the base is 10. Note how the powers of 10 increase by 1, each time we move left in the digits.

When we write a number in binary (base 2), we use 2 digits: 0 and 1. In writing the number **1001** , we mean the value

$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 1 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1,$
which has a value of 9 in base 10.

For notation, we may write $\mathbf{1001}_2$ to be explicit that the base is 2. Note how the powers of 2 increase by 1, each time we move left in the digits.

Binary

Just to see the pattern, we can count in binary from 0 to 10 as follows:

0
1
10
11
100
101
110
111
1000
1001
1010
...

Binary

To write a number N in binary, we find the largest power of 2 that does not exceed N, setting the largest digit. Then, we subtract this power of 2 from N and try to find the largest power of 2 that does not exceed the remainder, fixing a next digit, and so on:

To write $\mathbf{9}_{10}$ in binary, we note that $2^3 = 8 < \mathbf{9} < 2^4 = 16$. So we use 1×2^3.
The remainder is $9 - 8 = 1$.
Since both $2^1 = 2$ and $2^2 = 4$ exceed $\mathbf{1}$, we must have contributions of 0×2^2 and 0×2^1.
Then, since $\mathbf{1} = 2^0$, we use 1×2^0.

This gives us 1001 as we had already seen.

Binary

Thus, just using a series of 0's and 1's, we can express positive integers.

We can add binary numbers just like decimal numbers, where $1 + 1 = 10, 1 + 1 + 1 = 11$:

```
   111
  111011
+ 111110
--------
 1011001
```

Binary

Subtraction can actually be handled by addition using something called the **two's complement** (or other methods).

Negative integers can be similarly represented, but we will not discuss these methods.

Fractional values can be represented in binary, too, where one bit tracks the sign, some bits track the fractional part, and some bits track the exponent. Consider, for example:

$$\overbrace{(-1)^{0 \text{ or } 1}}^{\text{sign}} \overbrace{b_0.b_1 b_2 b_3 \ldots b_n}^{\text{n+1 bit mantissa}} \times \overbrace{2^{B_1 B_2 B_3 \ldots B_N}}^{\text{N bit exponent}}$$

where each of $b_0, b_1, \ldots b_n$ and $B_1, B_2, \ldots B_N$ are 0 or 1.

We can then have scientific notation and fractional values in binary. For example,

$$-1.0101001 \times 2^{101_2}.$$

Numeric Types

Depending on the level of precision desired, whether or not the sign matters (for integer types), and whether we wish to store fractional values, we can appropriately choose a type of data.

The **sizeof** a variable or data type tells us how many bytes are assigned to it.

Obviously the larger the size, the wider the range of values and/or the greater the precision a given data type can afford. We can never be infinitely precise for fractional numbers and we can never cover all possible integers because we only have a finite amount of storage for our data!

Numeric Types

To represent **unsigned** integer types (≥ 0), some common types are **unsigned char**, **unsigned short**, **unsigned int**, **unsigned long int**, and **unsigned long long int**.

In general, the sizes in **weakly increasing order** (i.e. a value to the right is bigger than or equal to the value on the left) are **sizeof(unsigned signed char), sizeof(unsigned short), sizeof(unsigned int), sizeof(unsigned long int),** and **sizeof(unsigned long long int).**

Numeric Types

For example, it may be the case that **sizeof(unsigned char) == 1**, **sizeof(unsigned short) == 2**, **sizeof(unsigned int) == 4**, **sizeof(unsigned long int) == 4**, and **sizeof(unsigned long long int) == 8**.

The use of $==$ in lieu of $=$ is intentional: this is consistent with C++ syntax for these comparison operators.

Numeric Types

As an example with precision, we cannot express any number larger than 511 with an **unsigned char**.

In code, we can abbreviate **unsigned int** as **unsigned**; **unsigned long int** as **unsgined long**; and **unsigned long long int** as **unsigned long long**.

Numeric Types

To represent **signed** integer types, some common types are **signed char**, **short**, **int**, **long int**, and **long long int**.

In general, the sizes in weakly increasing order are: **sizeof(signed char), sizeof(short), sizeof(int), sizeof(long int),** and **sizeof(long long int)**.

For example, we may have **sizeof(signed char) == 1**, **sizeof(short) == 2**, **sizeof(int) == 4**, **sizeof(long int) == 4**, and **sizeof(long long int) == 8**.

Numeric Types

Often, unless the range of data is very large, **int** is a good data type to represent a signed integer type, and **unsigned int** is a good data type to represent a generic unsigned integer type.

Assuming **sizeof(int)** and **sizeof(unsigned int)** are both 4 and **sizeof(long long int)** and **sizeof(unsigned long long int)** are both 8, with 8 bits to a byte,

the range of **int** is roughly from -2 billion to $+2$ billion,

the range of **unsigned int** is roughly from 0 to $+4$ billion,

the range of **long long int** is roughly from -10^{19} to 10^{19} and

the range of **unsigned long long int** is roughly from 0 to 2×10^{19}.

Numeric Types

To manage numbers with a decimal, we use **floating point numbers**: one of **float**, **double**, and **long double**. All floating points are signed.

In weakly increasing order of sizes, we have **sizeof(float)**, **sizeof(double)**, and **sizeof(long double)**.

All floating point numbers are signed. Typically **double** is a good all-purpose floating point number, usually occupying 8 bytes where 1 bit is used to record the sign $\pm$, 52 bits are used for the mantissa, and 11 bits are used for the exponent.

A **double** gives approximately 15 digits of precision, and a range of values from -10^{308} to 10^{308}.

Numeric Types

A **bool** is an integer type that can store **true** (with a bit pattern representing 1) and **false** (with a bit pattern representing 0).

A **char** is an integer type that represents each character of the keyboard (and the white-space and some non-printing characters) as an integer value. Typically this value is the ASCII value of the character. For example, all of
'.', 'A', 'z', ' ', '\'', and '\n'
can be represented as a type **char**.

Note that **char** is not necessarily the same as **signed char** or **unsigned char**. But the C++ standard guarantees that **sizeof(char) == sizeof(signed char) == sizeof(unsigned char) == 1**.

Numeric Types

By default, when we write an **integer literal** (i.e. no decimal appearing), it is taken to be of type **int**; and by default, if we write a number with a decimal, a **floating literal**, it is taken to be of type **double**.

```
13; // this 13 is of type int
4.; // this 4. is of type double
1.06; // this 1.06 is of type double
```

Numeric Types

We can change the default interpretations of numeric literals by adding a suffix.

L or **l** represent **long**;
U and **u** represent **unsigned**; and
F and **f** represent **float**:

```
13f; // this 13 is of type float
13ull; // this 13 is of type unsigned long long int
4.4L; // this 4.4 is of type long double
```

Numeric Types

C++ also supports scientific notation for floating point numbers. We can, for example represent 1.23×10^{10} or -2.46×10^{-20} by

```
1.23E+10;
-2.46E-20;
```

and actually **e** can be used in lieu of **E**.

Arithmetic Rules

We have seen that mathematical computations can be done, for example:

```
cout << 4.4 + 2.1 - 1/(6-2);
```

There are some important rules to be mindful of. Order of operations is obeyed so that 4.4 and 2.1 are added, and the value of 1/(6-2) = 1/4 is computed and subtracted from the sum...

Pairing of integers: the result of any +, -, * or / between two integer types will be an integer type! When a result, such as in division, yields a fractional value, the decimal part is truncated if the result is to be an integer type.

Pairing with a floating point: the result of any +, -, *, or / between any two numeric types where at least one of them is a floating point will be a floating point. Thus, we can add 4.4 + 2.1 + 0 and arrive at 6.5 instead of 6.

Arithmetic Rules

Left to right evaluation: when operations of the same precedence appear, + and -, say, the result is evaluated left-to-right. For example 1 + 2 + 3 becomes (1+2) + 3 = 3 + 3 = 6.

The line of code will output:

```
6.5
```

4.4 + 2.1 = 6.5 because **both** 4.4 and 2.1 are of type **double**.

6-2 = 4: **both** 6 and 2 are **int**s, so 4 is an **int**.

1/4 = 0: **both** 1 and 4 are **int**s, so 0.25 gets truncated to 0, an **int**.

6.5 - 0 = 6.5 because 6.5 is **double**, which has an **int** 0 subtracted from it, and the result should be a **double**.

Program with User Inputs I

We will consider two different versions of a program that compute the average of 3 numbers given by the user.

Average1.cpp

```
#include<iostream>

using namespace std;

int main() {
    int value1 = 0; // first value
    int value2 = 0; // second value
    int value3 = 0; // third value
    int sum = 0; // to store the sum

    cout << "Please enter a number: "; // prompt first
    cin >> value1; // read in their first number
```

Program with User Inputs II

```
   cout <<"Please enter another: "; // prompt for second
   cin >> value2; // read in their second number

   cout <<"Please enter another: "; // prompt for third
   cin >> value3; // read in their third number

   // compute the sum of the numbers
   sum = value1 + value2 + value3;

   // find the average and display it: average is sum/3
   cout << "The average is: "<< sum/3;

   return 0;
}
```

Program with User Inputs III

```
Please enter a number: 7
Please enter another: 7
Please enter another: 0
The average is: 4
```

Program with User Inputs

We will analyze this first program line by line:

```
int value1 = 0;
```

This creates an **int** variable called **value1** with an initial value of **0.** This statement **defines** the variable in that hereafter, **value1** refers to a specific location in memory that represents an **int.** The value stored in the memory location of **value1** can change, but it will always be an **int** in the same location of memory.

The same idea applies to the next 3 statements

```
int value2 = 0;
int value3 = 0;
int sum = 0;
```

Program with User Inputs

```
cin >> value1;
```

We use the console input variable, **cin** to read a user's input, which is parsed when the user hits [ENTER].

This is an instruction to accept the next input the user provides, which should be an integer value, and overwrite the 0 stored in **value1;**

The same idea applies to

```
   cin >> value2;
```

and

```
   cin >> value3;
```

Program with User Inputs

```
sum = value1 + value2 + value3;
```

We compute the sum of **value1**, **value2**, and **value3** and use this value to overwrite the value stored in **sum**.

The **=** is the **assignment operator**. It does not represent equality, but rather means to overwrite the value on the left side with what is on the right side.

Unlike in many sciences where a variable may be constant, it's important to recognize that in programming, a variable can be updated!

Program with User Inputs

```
cout << "The average is: "<< sum/3;
```

Due to integer division, we do not get the value of 14/3 = 4.666667.

If we instead had defined **sum** to be of type **double** from the beginning, then the average would be computed as 4.666667.

Program with User Inputs

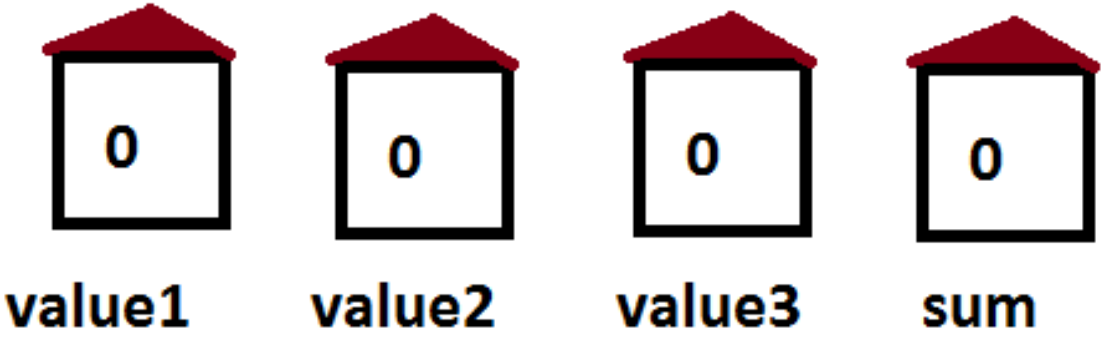
We look at the memory allocation and values for the different variables just after they are all defined and just before the average is printed.

Each variable has its own “home” in computer memory. The values can be updated through the running of the program.

Program with User Inputs

We look at the memory allocation and values for the different variables just after they are all defined and just before the average is printed.

Each variable has its own “home” in computer memory. The values can be updated through the running of the program.

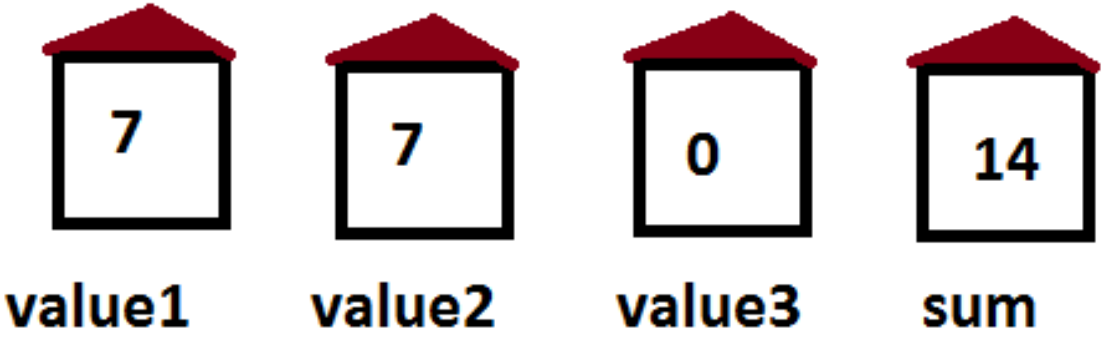



```
int value1 = 0;
int value2 = 0;
int value3 = 0;
int sum = 0;
```

Program with User Inputs

We look at the memory allocation and values for the different variables just after they are all defined and just before the average is printed.

Each variable has its own “home” in computer memory. The values can be updated through the running of the program.




```
//...
cout << "The average
is " << sum/3;
```

Program with User Inputs

Some further remarks:

- Variables (and most entities) in C++ can only be defined once throughout an entire program. Therefore, the following snippet would be in error:

```
int x;
double y = 7;
int x = 11; // ERROR: x is already defined!
char y = '.'; // ERROR: y is already defined!
```

  Even if the variable is re-defined to be of the same type, this is in error. The variable already had a place in memory.

- In a like manner to integer division, if a floating point value is stored in an integer type, the decimal part is removed:

```
int x = 4.99999; // x is 4
x = 2.7; // x is now 2
```

Program with User Inputs

- Valid variable names may...
  - begin with any letter, lowercase or uppercase (a-z, A-Z), or with an underscore _ and then
  - contain any number of additional chars that are letters, lowercase or upper case, digits from 0-9, or underscores _.

  A name that does not adhere to these standards is invalid.

```
int __AN_iNt999 = 4; // okay
double D0uBL3 = 3.14; // okay

int 3x = 14; // ERROR: beginning with digit
double ?variable = 0.6; // ERROR: uses a ?
```

Program with User Inputs

- Variable names should be descriptive. Through appropriate names of variables (and later functions and classes), code should almost be self-documenting, although comments are still required!

  Calling the variables **a**, **b**, and **c**, instead of **value1**, **value2**, and **value3**, would make the code less clear to read.

- There are two common conventions for variable names.

  Camel case:

  ```
  double aVariableNameLikeThis;
  int OrLikeThis;
  ```

  Using underscores:

  ```
  double a_variable_like_this_uses_underscores;
  ```

Program with User Inputs

- It is also good practice to name variables to avoid “magic numbers” - numbers/parameters that appear in the code but are otherwise unclear as to what they are exactly. Compare below:

```
double subTotal = 14.33; // dollars
double taxRate = 0.10; // 10% tax

// calculate total with taxRate included
double grandTotal = subTotal * (1 + taxRate);
```

with the below:

```
double subTotal = 14.33; // dollars

// What is 1.1? Why is it here?
double grandTotal = 1.1 * subTotal;
```

Program with User Inputs I

Here is a different version of a program that compute the average of 3 numbers given by the user.

Average2.cpp

```
#include<iostream>

using namespace std;

int main() {
    // the values the user will input
    int value1, value2, value3;

    // read in all 3 numbers
    cout << "Please enter 3 numbers separated by spaces: ";
    cin >> value1 >> value2 >> value3;
```

Program with User Inputs II

```
    // compute the sum of the numbers
    int sum = value1 + value2 + value3;

    // find the average
    double floatingAverage = static_cast<double>(sum)/3;

    cout << "The average is: "<<floatingAverage;

    return 0;
}
```

```
Please enter 3 numbers separated by spaces: 7 7 0
The average is: 4.66667
```

Program with User Inputs

We'll now look at some of the lines of this program.

```
int value1, value2, value3;
```

This defines (gives them a place in memory) the **int** values **value1**, **value2**, and **value3**, but leaves them **uninitialized**.

If a value is **uninitialized**, then its initial value is whatever bit pattern happens to be at that memory location - this could vary from computer to computer, even on different runs on the same computer.

Program with User Inputs

Multiple variables of the same type can be defined and/or initialized on a single line, by separating the variables by commas. For example,

```
double w, x = 4, y, z = 6;
```

defines **w, x, y**, and **z**, but only assigns a specified value to **x** and **z.**

It is generally safer to initialize all variables, in case there are errors in the program that prevent them being set properly. Otherwise the program can have unpredictable behaviour, sometimes working, sometimes not, depending on the value of the uninitialized variables.

Program with User Inputs

```
cin >> value1 >> value2 >> value3;
```

Just like **cout**, **cin** can form chains of input with **>>**.

This will read in three integer values from the user, separated by white space, and set the values of **value1**, **value2**, and **value3**.

When using **cin >> ...**, there are a few rules for parsing.

First, whitespace - spaces, tabs, new lines - is ignored until a non-whitespace character is found. Then, **cin** will only read in data until it either encounters another whitespace character or when the data “stops making sense.”

Program with User Inputs

```
cin >> value1 >> value2 >> value3;
```

By the user entering **7[SPACE]7[SPACE]0[ENTER]**:

- **7** is read into **value1** and **cin** stops reading at the space; then
- **cin** ignores the space and reads **7** into **value2**, stopping at the space; then
- **cin** ignores the space and reads **0** into **value3**, stopping at the new line.

Yes, hitting [ENTER] generates a **n** character.

Program with User Inputs

```
double floatingAverage = static_cast<double>( sum ) / 3;
```

To guard against integer division, we compute the value of **sum** and turn that into a **double** for the sake of the computation.

This **static_cast** has no effect upon the **sum** variable: it is still an **int**. But the value used in the computation is as though **sum** were of type **double**.

In general, if we wish for a given value/expression to be converted to **newType** for the sake of a computation, we use the expression

static_cast<newType>(expression)

Program with User Inputs

```
double floatingAverage = static_cast<double>( sum ) / 3;
```

The same effect could have been had by writing:

```
double floatingAverage = sum/3.;
double floatingAverage = sum/3.0;
double floatingAverage = (sum+0.0)/3;
```

Program with User Inputs

Note that

```
double floatingAverage = sum/3;
```

is valid code, but sum/3 is first evaluated as an int and although **floatingAverage** is of type **double**, it will just store the floating point equivalent of the integer division. The same idea holds true for

```
int x = 17;
double y = x; // y is a double, but it stores 17
```

Program with User Inputs

Further remarks on casting:

Sometimes it is ueful to cast to an **int** to achieve rounding. Given a floating point number **num**,

- When **num** is positive, **static_cast<int>(num + 0.5)** is **num** rounded to the nearest integer.
- When **num** is negative, **static_cast<int>(num - 0.5)** is **num** rounded to the nearest integer.

For example, with 1.43: 1.43+0.5 = 1.93, which truncates to 1, the correct rounded value. And with -0.98: -0.98 - 0.5 = -1.48, which truncates to -1, the correct rounded value.

Note: it is not good enough to just use **static_cast<int>(num)** as, for example, -0.98 would become 0 instead of -1.

Program with User Inputs

It is important to be clear in written code. There is an older C-style cast syntax that in these cases could achieve the same result:

```
double floatingAverage = ( (double)(sum) ) / 3; // bad style
```

but this is ill-advised as such syntax can actually refer to a few different types of casts. Someone reading the code quickly might not realize what the intent of the cast is. Use **static_cast** for clarity.

Arithmetic Shorthand

C++ allows us some shorthand for updating variables. In each grouping below, the statements all do the same thing. The variables **val1** and **val2** can be *any numeric type:* **int**, **long double**, **unsigned int**, etc.

```
val1 = val1 + val2; // overwrite val1 by the value of val1 + val2
val1 += val2;

val1 = val1 - val2; // overwrite val1 by the value of val1 - val2
val1 -= val2;

val1 = val1 * val2; // overwrite val1 by the value of val1 * val2
val1 *= val2;

val1 = val1 / val2; // overwrite val1 by the value of val1 / val2;
val1 /= val2;
```

In all cases, the type of **val1** remains unchanged. Even if **val1** is an integer type and **val2** is a floating point, **val1** may only store an integer.

Arithmetic Shorthand

Modular arithmetic can be performed using the % operator. The **operands** (arguments used) should be integer types.

Recall sometimes integers don't divide evenly into each other:

$$\frac{14}{3} = 4 \quad \text{with a remainder of 2.}$$

And integer division tells us that **14/3 = 4**.

Arithmetic Shorthand

We could find the remainder via **14 - 3 * (14/3)** since this is **14 - 3*4 = 2**. But we can also code something such as:

```
cout <<14 % 3; // will print 2
```

Similar to +, -, *, and /, if **val1** and **val2** variables then the two statements below are equivalent

```
val1 = val1 % val2;
val1 %= val2;
```

Arithmetic Shorthand

There are also two other operators **++** and **- -**, each with two different forms (a prefix and postfix).

Prefix: writing

```
++val;
```

will increment the value stored in **val** by 1 and in addition, the overall value of the expression **++val** will be the new value of **val**.

Postfix: writing

```
val++;
```

will increment the value stored in **val** by 1 and in addition, overall value of the expression **val++** will be *the previous value* of **val**.

Arithmetic Shorthand

```
int val = 7;

cout << "val++ is: "<< (val++) << endl; // now val is 8

cout <<"++val is: "<< (++val); // now val is 9

cout <<"val is: "<<val;
```

```
val++ is 7
++val is 9
val is 9
```

The same holds for the **- -** cases, except the values are decremented by 1. The prefix version evaluates to the new value and the postfix version evaluates to the old value.

Overflow, Underflow, and Rounding Errors

Because integer and floating point types only have a finite number of bits to store values, we can have some odd behaviours:

```
cout << 1000000*1000000 << endl; // somehow negative!

cout <<2u*3000000000u << endl; // definitely not 6000000000

cout <<1u - 5 << endl; // not -4

double seventh = 1./7;
double fortyNinth = seventh/7;
cout << 49*fortyNinth - 1. <<std::endl; // not = 0!!!

cout <<10*1.E+308; // equals infinity...?
```

```
-727379968
1705032704
4294967292
-1.11022e-16
inf
```

Overflow, Underflow, and Rounding Errors

In the first case, **1000000** is taken to be an **int**: the multiplication is carried out, but with perhaps only 4 bytes of information, the resulting bit pattern is interpreted as a negative number. When signed integer types go above or below their maximum or minimum possible values, we have **undefined behaviour** (it could vary from one compiler to another).

An **int** can only go as large as **INT_MAX** and as low as **INT_MIN** before overflowing or underflowing and causing this undefined behaviour.

The values such as **INT_MAX** and **INT_MIN** are system-dependent, as is what happens to a value when overflow or underflow happens, hence the term "undefined", since anything can happen when underflow/overflow occurs.

Overflow, Underflow, and Rounding Errors

In the second case, **6000000000** exceeds the max possible value of an **unsigned int** so the value “wraps” around. After the largest value is obtained, it starts counting again at 0.

Likewise, in the third case, by dropping below 0, the **-4** gets wrapped around and the resulting number is 4 values less than the max possible **unsigned int** value.

This wrapping around for unsigned integer types will always happen.

Just imagine an unsigned integer type that can only go as large as 7: counting up, we would have 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, ...; and counting down we would have 7, 6, 5, 4, 3, 2, 1, 0, 7, 6, 5, ...

Overflow, Underflow, and Rounding Errors

Because of a limit in the precision of **double**, the value of **fortyNinth** was not exactly equal to 1/49 and we see there is a small error on the order of 10^{-16}. Floating point numbers are subject to this sort of **rounding error.**

In the latter case, we went above the max possible value of a **double** and got the value **inf**.

Strings

A **std::string** is a class type defined in the C++ Standard, in the **std** namespace, that makes managing strings of text very easy.

A **std::string** is a **container** (a data structure that stores stuff) that stores **char**s.

The **std::string** class is defined within the **<string>** header.

Strings

Here are some basic functions of the class:

- Creating an empty **std::string**:

  ```
  string s1; // s1 is a string storing ""
  ```

- Changing the value of a **std::string** by using the assignment operator **=** and a string literal:

  ```
  s1 = "Ahoy!"; // now s1 stores "Ahoy"
  ```

Strings

- Creating a **std::string** with an initial value:

```
string s2( "not empty"); // s2 is a string storing " not empty"

string s3 = "also not empty"; // s3 is a string storing " also not empty"
```

- Changing the value of a **std::string** by using the assignment operator **=** and another **std::string**:

```
s1 = s2; // now s1 is a string storing "not empty"
```

- Displaying a **std::string** to the console:

```
cout << s1 << endl << s3 << endl;
```

```
not empty
also not empty
```

Strings

- Concatenation with **+**:

```
cout << s1 + s3 << endl;

/* make a string by prepending the char '@' to s1 and appending
   "... " the contents of s3, and the char '!'*/
string s4 = '@'+ s1 + "... "+ s3 + ' !';

cout << s4 <<endl;
```

```
not emptyalso not empty
@not empty...  also not empty!
```

A **std::string** can be concatenated with other **std::string**s, or with a string literal or a **char**. The **+** operator "adds" the strings together to produce a new string. At least one side of the **+** has to be a **std::string**. We **cannot add string literals:**

```
cout <<"hello"+ " world!"; // ERROR: both are string literals
```

Strings

- If we concatenate with **+=**, the left-hand side is modified:

```
cout << s1 <<endl;
s1 += s3; // append s3 to s1
cout <<s1; // now s1 is changed!
```

```
not empty
not emptyalso not empty
```

- Finding the length of a **std::string**:

```
s1 = "i like food";

// store the number of chars in s1 in variable called s1Length
size_t s1Length = s1.size();
cout <<s1Length; // outputs 11
```

Variables of type **size_t** are especially important for containers.

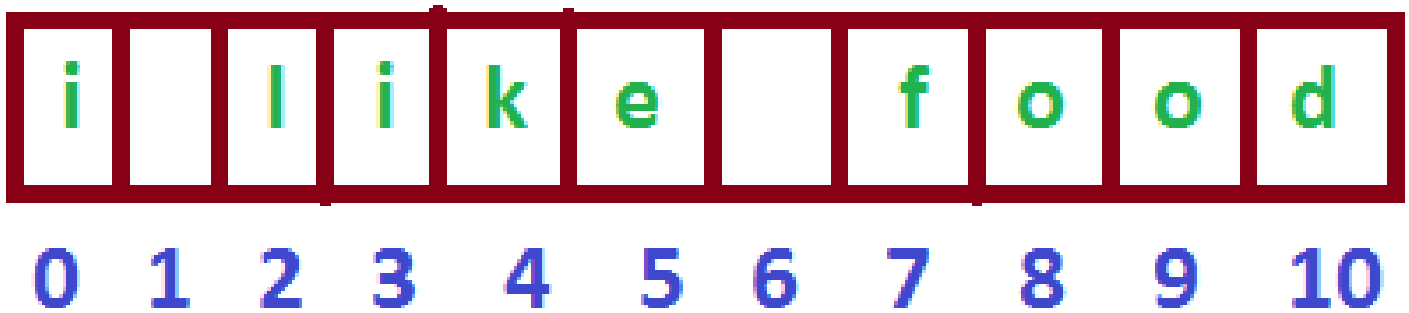
Strings

- **std::string**s index their characters from 0. We can access or modify individual characters.

```
char first = s1[0]; // first is a copy of the first char of s1: 'i'
size_t fifthIndexValue = 4;
cout << first <<s1[fifthIndexValue] << endl;
```

```
ik
```

We call the square brackets **[]** the **subscripting operator**. Note that the largest index of a **string** that we can access is its size minus 1. Above, **s1.size()-1** is **10** is the largest permissible index because we started counting at index 0!



Strings

The variable type **size_t** is an unsigned integer type that *should be used whenever we are talking about the size of a container and whenever we wish to index a container.* Depending on the computer architecture, **size_t** may differ: could be **unsigned int**, **unsigned long long**, etc.

A variable of type **size_t** is the smallest (in memory usage) unsigned integer type that is guaranteed to be able to access every element of a container.

Just imagine: if a **std::string** stored 14 billion elements and we wrote something like:

```
int stringSize = theString.size(); // VERY BAD
unsigned int stringSize2 = theString.size(); // ALSO VERY BAD
```

Then because **int**s can only go up to about 2 billion and **unsigned int**s can only go up to about 4 billion, we would not be correctly storing the size of the string.

Strings

- The **subscript operator** gives us direct access to modify the **char**s:

  ```
  s1[0] = 'I'; // change first char to 'I', now s1 stores "I like food"
  ```

- We can add a character at the end with **push_back**:

  ```
  s1.push_back('.'); // now s1 stores "I like food."
  ```

  Note that we have to specify a **char** to insert inside the parentheses.

- We can also remove the final character with **pop_back**:

  ```
  s1.pop_back(); // now s1 stores "I like food"
  ```

Strings

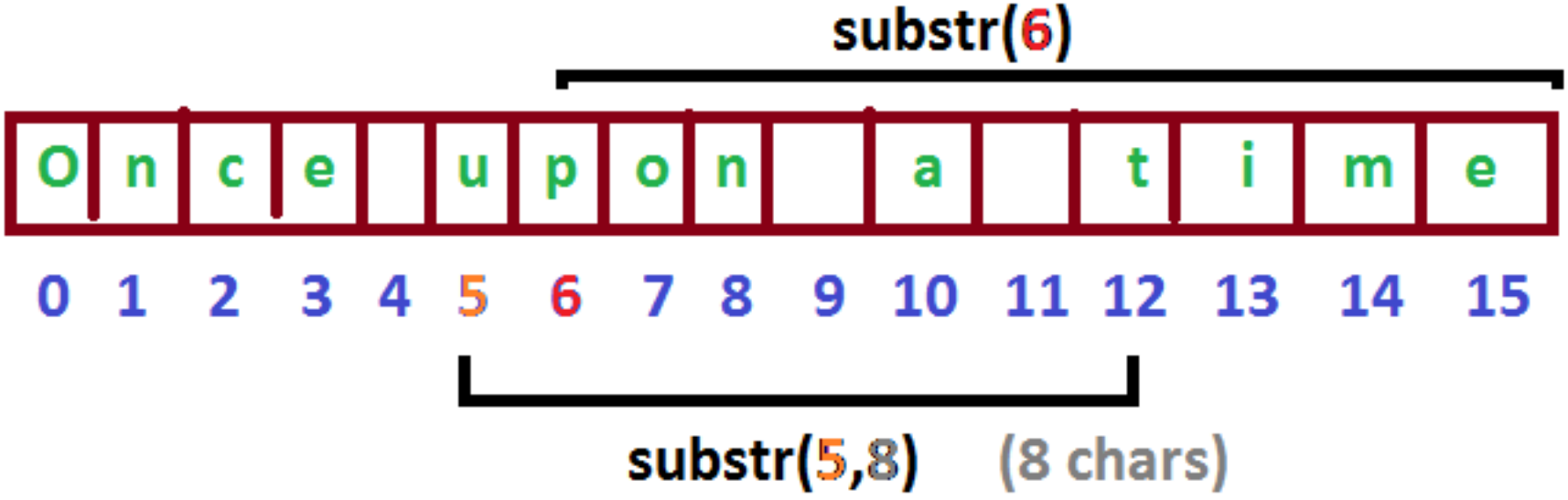
- Extracting substrings:

```
s1 = "Once upon a time";

// print substring of length 6 beginning at index 5
cout << s1.substr(5, 8) << endl;

// overwrite s1 with substring from position 6 to end
s1 = s1.substr(6);
cout << s1;
```

```
upon a t
pon a time
```

The **substr** function can be given 1 or 2 **arguments** (inputs). With one argument, we extract a substring from that index value to the end; with two arguments, we extract a substring with size given by the second argument, beginning at the index of the first argument.

Strings


substr(6)
O n c e u p o n a t i m e
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
substr(5,8) (8 chars)

Strings

- **std::string**s can be compared **lexicographically** (like in a dictionary), based on the **char**s that comprise them.

  The following is a list of how some of the individual **char**s compare:

  [EMPTY STRING] < ' '<
  '0'< '1'< '2'< ... < '8'< '9'<
  'A'< 'B'< ... < 'Y'< 'Z'<
  '_'< 'a'< 'b'< ... < 'y'< 'z'

  When we compare two **std::string**s, the result is either **true** or **false**.

```
string e1 = "Ze", e2 = "a", e3 = "Z7q", e4 = "Z7q";

e1 < e2; // is true because 'Z'< 'a'
e3 < e1; // is true because the 'Z' s match, but '7'< 'e'
e3 == e4; // is true because the strings match in every character
```

Strings

Note that equality is denoted by **==**, not **=** because the latter is the assignment operator!

- We can search for a substring within a string:

```
string numbers = "one, two, three, four, five, six, seven, eight, nine,
    ten";

// will be found
size_t position1 = numbers.find("en");

// will not be found
size_t position2 = numbers.find("qqqqqqqqqqq");
```

Strings

If the substring is found, an index is returned to the first place in the string where it appears. If the substring is not found, the **find** function returns the special value **std::string::npos**.

Thus, **position1** will be 37, the index of the second 'e' of "seven" and **position2** will be **std::string::npos**.

As an example of checking if an item is not found in a string, we would need to check a condition like below:

```
if ( position2 == string::npos ) { // check if nothing found
    cout << "Not found in list";
}
```

string::npos evaluates to static_cast<size_t>(-1), some ginormous number that exceeds the largest possible index of a **std::string**.

Strings

- There are also other ways to construct **std::string**s:

  ```
  string repeatA( 5, 'A'); // repeatA will store "AAAAA"

  string repeatA2 = repeatA; // repeatA2 will be a copy of repeatA

  string repeatA3 ( repeatA ); // repeatA3 will be a copy of repeatA
  ```

  In the first case, within a set of parentheses, we can give two arguments: the first is an initial size for the **std::string** and the second is a character to repeat that many times.

- We can easily clear all their contents:

  ```
  repeatA.clear(); // now repeatA will store ""
  ```

Strings

- A **std::string** is constructed by putting appropriate parameters in parentheses after the class name:

```
/* this created an empty string with no name that no longer exists
   after this single line */
string();

/* creates a string storing "word", which is displayed, and then
   ceases to exist */
cout << string("word") << endl;

/* creates a string storing "C++", which then has the size function
   called for printing, then ceases to exist */
cout << string("C++").size();
```

```
word
3
```

Getline and Buffering

User input is stored in a buffer: the list of all inputs that have yet to be processed. These inputs are taken in when the user hits [ENTER] (and this also adds a new line character).

Through the operator **>>**, if ever **cin** is unable to make sense of its input then it enters a failed state. In a failed state, it will not modify further variables and it will not read any more user input.

Getline and Buffering

Consider the snippet

```
char c;
int i = 0;
double d;
double d2 = 4.38;
cin >> c >> i >> d >>d2 >>c;
```

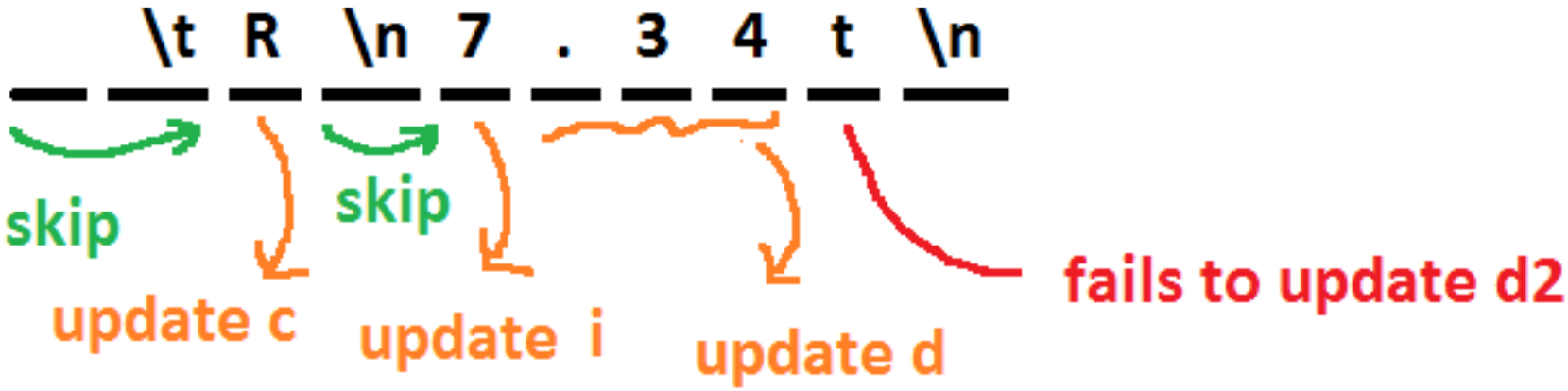
Getline and Buffering

cin >> c >> i >> d >>d2 >>c;

[space][tab]R[ENTER]7.34t[ENTER]...

- The whitespace (both space and tab) is skipped, **'R'** is read into **c** and **cin** stops because it only needed to read one character. The **new line** is left in the buffer.
- **cin** then needs to read a value for **i** and it skips over the new line and waits for more input to be added to the buffer (the user needs to enter more).
- **cin** reads the value **7** and uses this to update **i**, but it leaves the **.** and everything else in the buffer because a **.** doesn't make sense for an **int**.
- **cin** reads the value **.34** and uses this to update **d**, but leaves the **t** and new line in the buffer because **t** does not make sense for **double**.
- **cin** is unable to make use of **t** to update the **double** value **d2** so it enters a failed state. This leaves **d2** unchanged with a value of **4.38**.
- Because **cin** has failed, it will not even try to use anything else in the stream and **c** is left as **'R'**.

Getline and Buffering



Getline and Buffering

The operator **>>** isn't particularly useful for collecting entire lines of text because readings stop at white space.

```
// ask an important question
cout << "Why are cats so awesome? ";

string reason; // the answer
cin >> reason; // read in their reason

cout << "Your answer was: "<< reason;
```

Had the user given an entry of **They just are.** then the output of the code would be

```
Why are cats so awesome?  They just are.
Your answer was:  They
```

Only the first word is collected.

Getline and Buffering

To fix this, we use the **getline** function.

It takes two arguments: an input stream like **cin** and a string variable to set.

The function reads from the buffer **up to, but not including the first new line character that it comes across**. It uses this reading, **which is the empty string if the first character it sees is a new line**, to set the string. It then advances past the new line character but does no further reading.

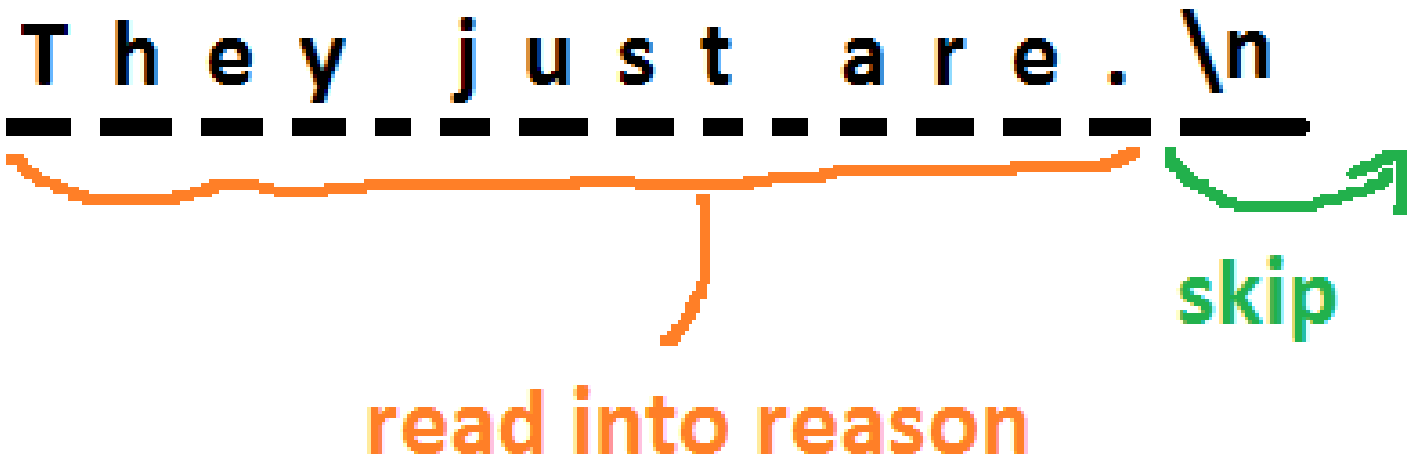
Getline and Buffering

Given the same user input, a functional version of the previous code would be

```
// ask an important question
cout << "Why are cats so awesome? ";

string reason; // the answer
getline(cin, reason); // read in the entire input line for the reason

cout << "Your answer was: "<< reason;
```

```
Why are cats so awesome?  They just are.
Your answer was:  They just are.
```

Getline and Buffering



Getline and Buffering

To clear a possible new line character that remains in the buffer after a reading using **>>**, we often extract the next **char** from the buffer. To do this, we can write either

```
cin.get();
```

or

```
cin.ignore();
```

The difference between the two is that **cin.get()** returns the value of the next **char** before moving forward in the stream, whereas **cin.ignore()** just moves forward in the stream. Any valid char, including new lines, tabs, etc. are counted here: these functions do not skip white space.

Getline and Buffering

```
double d = 0;
string s;

// ... code and such

cin >> d;

cin.ignore(); // moves past new line char
getline(cin, s); // reads a full line of text without just setting s to be empty
```

Getline and Buffering

Remarks: When use input is accepted and the user presses [ENTER], a new line is generated in the console window and any subsequent displays appear on the next line. Otherwise, a new line is not generated unless we use a new line character, or **endl**, etc.

It's very easy to forget to skip over a new line character that remains in the buffer after **>>** before using **getline**.

Math

In C++, we can work with mathematical functions, but we need to include the **<cmath>** header. A table of some common math functions, along with their C++ signatures and implementations, is below:

In Math	C++ Function Signature	C++ Syntax
$\sin x$	**double sin(double);**	sin(x)
$\cos x$	**double cos(double);**	cos(x)
$\tan x$	**double tan(double);**	tan(x)
$\sqrt{x}$	**double sqrt(double);**	sqrt(x)
$\lvert x \rvert$	**double fabs(double);**	fabs(x)
$\min(x, y)$	**double min(double, double);**	fmin(x,y)
$\max(x, y)$	**double fmax(double, double);**	fmax(x,y)
x^y	**double pow(double, double);**	pow(x,y)
e^x	**double exp(double);**	exp(x)
$\ln x$	**double log(double);**	log(x)
$\log_{10} x$	**double log10(double);**	log10(x)

Math

A **function signature** lists its inputs (separated by commas, inside of parentheses) and its output type (before the function name).

double pow(double, double): the function named **pow** accepts two inputs of type **double** (if they are **int**s then they are evaluated and promoted to **double**) and returns a variable of type **double**.

Writing math in C++ is rather one-dimensional. Instead of writing

$$z = \frac{x\sqrt{y^7+2} + x}{\ln 2 + 1},$$

we would have to write

```
z = ( pow ( x, sqrt( pow(y, 7) + 2 ) ) + x ) / ( log(2) + 1 );
```

where we had to use lots of parentheses for order of operations, too.

Math

Bracketing can be a delicate art... It can be good practice to count +1 for each left bracket we come across (and -1 for each right bracket we come across). The overall sum should be 0 if done correctly.

```
      +1    +2       +3     +4  +3    +2 +1    0   +1  +2 +1     0
z = ( pow ( x, sqrt( pow(y, 7) + 2 ) ) + x ) / ( log(2) + 1 );
```

We also need to be careful to insert proper operations such as multiplication. Mathematically, we can write xy or $x(y-7)$, but in C++ we must write **x*y** and **x*(y-7)**, etc.

Formatting Output

By using the header **<iomanip>**, we can format how an output stream writes data.

To modify the default behaviours, we use **stream manipulators**.

The number of digits after the decimal for a floating point can be specified with an **int** 0 or greater along with using the **setprecision(int)** manipulator. This is generally done in conjuncation with specifying a **fixed** format (strictly displaying a given number of digits after the decimal point) without scientific notation or a **scientific** format (to force scientific notation).

The scientific vs fixed formatting can be undone by using an **unsetf** function with a parameter **ios_base::float_field**.

The current precision value can be accessed with the **precision()** function.

Formatting Output

```
int defaultPrecision = cout.precision(); // obtain default precision level

// display normally with defaults
cout << 3.14159265 <<endl;

// display with fixed format and setprecision of 3: 3 digits after .
cout << fixed << setprecision(3) <<3.14159265 << endl;

// display with scientific format and setprecision of 1: 1 digit after .
cout <<scientific <<setprecision(1) <<3.14159265 << endl;
```

```
3.14159
3.142
3.1e+00
```

Formatting Output

```
// undo the scientific and fixed formatting: back to normal display type
cout.unsetf(ios_base::floatfield);

// now try printing the number again
cout <<3.14159265 <<endl;

// go back to the old precision level, too!
cout <<setprecision(defaultPrecision);

// now back to normal!
cout <<3.14159265 <<endl;
```

```
3
3.14159
```

Remark: The precision value for the default (at least on Visual Studio) appears to include the number of digits before the decimal in the count of the precision.

Formatting Output

A stream can have a padding character specified by the **setfill** function. By default, this padding character is a space **' '**. In addition, it is possible to set the width of *the very next printed output* using the **setw** manipulator.

By default, the output is right-justified: for a given specified width, the last character of output is flushed against the “right margin.” But we can overwrite this so that the output is left-justified with the first character of output flushed against the “left margin.” To do this overriding, we use the **left** and **right** manipulators.

Formatting Output

```
// message to display
string message = "Hello";

// print it normally
cout << message <<endl;

// specify width of 10: default padding is ' ', right justified
cout << setw(10) << message << endl;

// specify fill char of '*' and set width to 11
cout << setfill('*') << setw(11) << message << endl;
```

```
Hello
     Hello
******Hello
```

Formatting Output

```
// the setfill char is still '*' and we now left justify with width of 11
cout << left << setw(11) <<message <<endl;

// setw must be set every time: no width specified, no padding appears
cout << message << endl;

// setw is ignored when the output is too large
cout << setw(3) << message <<endl;

/* combine with precision stuff: note that "(approximately)" takes
   up 15 chars */
cout <<"pi = "<< setprecision(2) << setfill('!') << setw(7) <<3.14159265
   << right << setfill(' ') << setw(16) << "(approximately)"<<endl;
```

```
Hello******
Hello
Hello
pi = 3.14!!! (approximately)
```

Formatting Output

Remarks: all of **left**, **right**, **setfill**, **setprecision**, **fixed**, and **scientific** have long-term effects in that they remain in effect until changed, **setw** only applies to the very next printed output.

If an output is wider than the width specified by **setw**, the output is displayed in full and **setw** is ignored.

Using **fixed** with **setprecision(0)** can display a floating point that is rounded to the nearest integer.

endl is also a manipulator, but we don't need to **#include<iomanip>** for it.

Function Documentation

Recall that a function has inputs (possibly none) and an output (or possibly not). Functions do useful things for us. Getting the size of a string or extracting a substring involved the use of functions. The getline function allowed us to update the value of a string through a stream, and so on.

The **signature** of a function tells us precisely what the output is, what the function name is, and what arguments it takes inside of parentheses.

If there is no output, we write **void**.

If there is no input, we put nothing inside the parentheses, and if there are multiple inputs, we separate them by commas.

Function Documentation

```
// input is an int, output is a bool
bool isEven(int);

// inputs are two doubles, output is a double
double pow(double, double);

// takes no input and returns an int
int rand();

// takes no input and returns nothing
void printHello();
```

Function Documentation

The signature of a function only gives limited information and a user of the functions may wish to know more. It is possible to label the input parameters. This allows us to write detailed comments in the **function declarations**.

A **function declaration** tells the compiler that a particular function exists, but does not provide its implementation details.

If a function is used, those details will have to be given somewhere, but they don't have to be given where the functions are declared, which makes code easier to read.

We consider some examples of well-documented functions. In general function documentations will begin with /** and terminate with */.

We denote each input with **@param nameOfInput** and the output, if there is one, is specified by **@return**.

Function Documentation

```
/**
   Determines whether a given integer is even

   @param toCheck the integer we are examining
   @return whether the integer is even: true if it is, false if it is not
*/
bool isEven(int toCheck);

/**
   Computes a base to an exponent

   @param base the base value
   @param exponent the exponent value
   @return the base raised to the exponent
*/
double pow(double base, double exponent);
```

Function Documentation

```
/**
   This function will return a random int value from 0 to RAND_MAX

   @return the random int value
*/
int rand();

/**
   This function will print the word "Hello" to the console with cout
*/
void printHello();
```

Classes and Interfaces

We need to build up some terminology for classes now.

- A **class** is a collection of variables and associated functions: a **std::string** stores a collection of **char**s along with its size; the class allows us to modify these characters or to easily access its size or find a substring or ...

  Or imagine an **Employee** class. Such a class could store an employee's name, employee ID, hourly wage. Useful functionalities could include accessing the name and ID, accessing or modifying the hourly wage, etc.

Classes and Interfaces

- A **class object** is a specific instance of a class. For example a **string** variable storing "Hello world!" or an **Employee** variable storing a name of "Joe Bruin", an ID of 123, and an hourly wage of 30.25 dollars per hour.

  A class is just a general data type, but a class object is that class idea put to use as a variable.

Classes and Interfaces

- A **constructor** is a special function that creates an instance of a class.

```
// calls a string constructor that accepts no arguments
string s;

// calls a string constructor that accepts a string literal
string s2("ABC");

// // calls a string constructor that accepts a size_t and a char
string s3(10,'A');
```

Note that when a constructor taking no argument is called, **we do not use any parentheses**

```
string s(); // INCORRECT!
```

Classes and Interfaces

- A **member variable** is a variable that belongs to a class. For example, an **Employee** class would likely have a **string** member variable to store the employee name, an **int** member variable to store the ID, and a **double** member variable to store the hourly wage.

- A **member function** is a function that belongs to a class. For example, **clear** and **substr** are member functions of the **std::string** class.

  When we wish to access a member of a class, we use the **. access operator**.

```
string s;
cout <<s.size(); // use . to access function size of the class
```

Classes and Interfaces

- An **accessor** function is a member function of a class that does not change the class' data. A **mutator** function is a member function of a class that can change the class' data.

```
string s = "The sunlight is too bright.";

/* substr is an ACCESSOR: the data of the class is unchanged by
   our retrieving a substring *
cout << s.substr(1,2);

/* clear is a MUTATOR: the class data is changed now - the string is
   empty! */
s.clear();
```

Classes and Interfaces

- Object oriented programming is build upon the idea of classes, and classes serve as a means of **encapsulation**.

  Encapsulation is a means to protect data from misuse in other parts of a program.

  For example, the size of a **std::string** would be protected from external modifications. It would be disastrous if a user of the **std::string** class could change the size of a **string** storing "Good morning" to, say, the value 3, without changing the characters.

  Another example: the ID of a given employee should not change, say, so while we should allow users of an **Employee** class know the ID, but they shouldn't be allowed to modify it arbitrarily.

  Most classes have a **public** interface: a collection of members that all users of the class can access, but there may be other parts of the class that no one can access or where access is limited.

Classes and Interfaces

When reading a class interface, it may look something like:

```
class NameOfClass {
public:
   // THIS IS THE PUBLIC PART OF THE INTERFACE
   // AND FOR NOW IT IS THE ONLY PART TO PAY ATTENTION TO

private:
   // STUFF TO BE IGNORED FOR NOW
};
```

At this point in the course, only the stuff following **public:** and before **private:** should be studied.

Classes and Interfaces I

This is an excerpt of what could be the **std::string** interface:

```
class string {
public:
   /**
      Class constructor: creates an empty string
   */
   string();

   /**
      Class constructor: create a string from a length and repeat
      character

      @param length the number of times to repeat a given character
      @param repeatCharacter the character to repeat
   */
   string( size_t length, char repeatCharacter );
```

Classes and Interfaces II

```
   /**
   Function will return a substring of the original string

   @param start the starting index of the substring
   @param length the number of chars in the substring
   @return the substring
   */
   string substr( size_t start, size_t length ) const;

   /**
   Function will clear the string and make it empty
   */
   void clear();

   // OTHER STUFF
};
```

Classes and Interfaces

Note that a class constructor has the same name as the class itself and a return type is not given.

In the preceding case, **string** is the name of the constructors. They are written as **string()** and **string(size_t length, char repeatCharacter)** and not **string string()** and **string string(size_t length, char repeatChar)**.

The **substr** function is an accessor and this is why we add the **const** keyword after it to tell the compiler the function does not modify the class.

The **clear** function is a mutator function and we do not add a **const** keyword after it.

Classes and Interfaces

Here's another example of a class interface:

```
class RandomInRange {
public:
   /**
   Constructor: creates a RandomInRange from an upper bound
   @param max an upper bound for random number generation
   */
   RandomInRange( int max );

   /**
   Function returns a random number from 0 to max inclusive
   @return a random int value between 0 and max inclusive
   */
   int getRandom() const;

   // OTHER STUFF
};
```

Header Files

A **header file** is a file that contains declarations of various functions and variables, along with class interfaces.

When someone just wants to know what functions are available, without looking at the nitty gritty implementations, they can look at the header file.

For example, the **RandomInRange** class interface could be given in a header file. A separate **.cpp** file giving the implementation details would be included in the work space, but a user of the class would not need to look at that implementation file.

Header Files

For someone to use the **RandomInRange** class, suppose its header file was called **RandomHeader.h**, then they would need to **#include** the header in their **.cpp** file and ensure the implementation file is in the same directory.

When we include a header from the Standard Library, we use angled brackets **< >** and don't give an extension.

```
#include<iostream>
```

When we include a custom written header file, we used quotation marks and include the **.h**:

```
#include "SomeHeaderName.h"
```

Header Files

Recall that it is often an error to define something more than once. To protect against this, every header file should begin with the lines

```
#ifndef __SOME_NAME__
#define __SOME_NAME__
```

and terminate with

```
#endif
```

When the preprocessor is including different header files, for each header file that it includes, it checks if a token called **__SOME_NAME__** is defined. If it is already defined then it skips over everything in the file and does not include that file. If the token is not already defined, it will define that token and insert the contents of the header file.

Since these header files often contain class definitions, these “header guards” protect against a class definition appearing more than once.

Header Files

The following 3 slides contain a simple use of a **Cat** class.

The file **Cat.h** defines the cat class and provides an interface for a user of the class.

The file **Main.cpp** contains the main routine and makes use of the class.

The file **Cat.cpp** just needs to be included in the work space but does not need to be understood at this point in time.

Placing these files in the same workspace and compiling and linking appropriately will generate a functional program.

Header Files

Cat.h

```
#ifndef __CAT__
#define __CAT__

#include<string>

class Cat {
public:
    /** Default constructor */
    Cat();

    /** function that returns cat meow
    @return the string "meow"
    */
    std::string talk() const;
};

#endif
```

Header Files

Main.cpp

```
#include<iostream>
#include "Cat.h"

using namespace std;

int main() {
    Cat fluffy; // create a Cat called fluffy

    cout << "Fluffy says: "<< fluffy.talk(); // output cat's meow

    cin.get(); // pause

    return 0; // terminate program
}
```

Header Files

Cat.cpp

```
#include "Cat.h"

// default constructor
Cat::Cat() { }

std::string Cat::talk() const {
    return "meow";
}
```

Classes and Interfaces

In header files, especially, it is important to not write **using namespace std;**. The reason is that any file that includes a header file with that line of could would have the standard namespace imposed upon it. This can lead to naming conflicts: maybe someone writes a class called **scientific**. A line of code such as

scientific x; // ERROR: ambiguous

is ambiguous. It is either a syntax error or it means to create an instance of the **scientific** class, assuming there is a constructor that takes no arguments...

For the same reason, **global variables**, variables that can be defined above **int main()** should generally be avoided because naming conflicts can arise.

The header file given, therefore, explicitly writes **std::string** instead of **string** because we need to specify the namespace directly.

Types and Value Categories

There are other types of variables than simply “plain vanilla” types.

Sometimes, we want a variable to be held constant, with its value unaltered. In this case, we can declare the variable as **const**.

This can be good coding practice as it can avoid unintentional modifications of parameters that shouldn’t be altered.

Types and Value Categories

```
const double gravity = 9.8; // m/s^2 - earth's gravitational CONSTANT

double distanceFallen = 0; // in m
double timeFallen = 0; // in s

cout <<"How long does the object fall for in seconds? ";

cin >> timeFallen; // accept input for how long the object falls

// formula for distance fallen is 1/2 * gravity * timeFallen^2
distanceFallen = 0.5*gravity*pow(timeFallen, 2);

cout <<"Over "<< timeFallen <<"s, an object will fall "<< distanceFallen
   << "m.";
```

```
How long does the object fall for in seconds? 3.3
Over 3.3s, an object will fall 53.361m.
```

Types and Value Categories

With the **const**ness, we are protected from an error such as:

```
const double gravity = 9.8; // m/s^2 - earth's gravitational CONSTANT

double distanceFallen = 0; // in m
double timeFallen = 0; // in s

cout <<"How long does the object fall for in seconds? ";

// ERROR: CANNOT CHANGE gravity!
cin >> gravity; // accept input for how long the object falls
```

If a variable is to be constant, we just define it with the syntax:

const plainValueType variableName = whatever;

In our example, the **type** of **gravity** is **const double**, whereas the **type**s of **timeFallen** and **distanceFallen** are **double**.

Types and Value Categories

An **lvalue reference** (we'll just call it a **reference**) gives us another name for a variable. A reference is always bound to the same location in memory. We cannot change what it references.

A **pointer** is a related idea, with a pointer effectively storing the location of a variable in memory and its type.

Both pointers and references are nice because they are "light weight": both of these data types effectively store an address, which is low on storage and easy to move around.

The general syntax of defining and initializing a reference to a variable of a type **valueType** is:

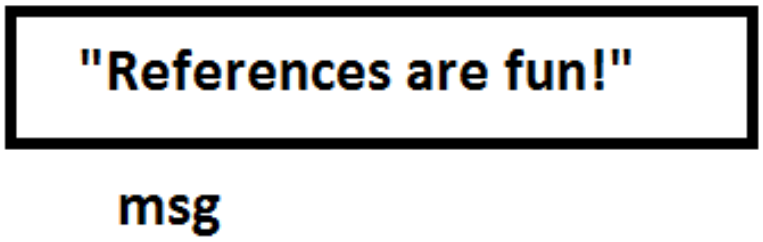
valueType & referenceName = someVariableToReference;

Types and Value Categories

```
// original message
string msg = "References are fun!";

// msgRef is just another name for msg
string & msgRef = msg;

// note how we can print msg by printing msgRef
cout << msgRef << endl;

// and we can change msg by changing msgRef
msgRef = "message changed";

// msg was changed
cout << msg << endl;
```

```
References are fun!
message changed
```

Types and Value Categories




```
string msg = "References are fun!";
```

Types and Value Categories

string &msgRef = msg;

msg (also known as msgRef)

Types and Value Categories

msgRef = "message changed";

msg (also known as msgRef)

Types and Value Categories

In a highly simplified view of the language, with a few elements left out, every expression that we look at in C++ has a **type** (including the possibility of **const**, being a pointer, or a reference) and a **value** category.

The **value** category will be either an **lvalue** or an **rvalue**.

Basically, something is an **lvalue** if it has an actual location in memory. Anything with a name ascribed to it is an **lvalue**. In particular, variables are lvalues, as are things that we can put on the left-side of an assignment operation.

Something is an **rvalue** if it is not given a place in memory, for example, the output of **someString.size()** is a value of type **size_t**, but that value is not stored in a permanent location in memory and it has no name.

```
string s; // is of of type string
string & sR = s; // sR is of type string&
```

Types and Value Categories

A reference can only be created from an lvalue and not an rvalue.

```
string s1 = "dragon";
string& s2 = s1; // okay: s2 references s1

char &c = s2[3]; // okay, s2[3] is same as s1[3] and has a place in memory

c = 'G'; // now s1 == "draGon"

/* ERROR: string("dungeon") is a string but it has no name - it's an
   rvalue */
string &s3 = string("dungeon");
```

Types and Value Categories

There are some subtleties to references. Although we cannot write

```
double &d = 4.7; // ERROR: cannot make reference to rvalue!
```

we can write

```
const double &d = 4.7; // totally okay because d is a reference to const
```

Above, **d** is considered a **reference to const double** and it is permissible for a reference to const to bind to an rvalue.

Types and Value Categories

Although we can write

```
const unsigned int u = 0;
const unsigned int & v = u;
```

because **v** is treated as a reference to **const unsigned int**, and hence can reference **u**, we cannot have

```
// ERROR: cannot make ordinary reference to a const value!
unsigned int& w = u;
```

because if we allowed that, we would be secretly giving permission to the program to modify the const value **u** through **w**, since **w** references **u.**

Types and Value Categories

We are allowed to use a reference to const to refer to a non-const value:

```
long double d = 1.010101;

// e references d, but we cannot change d through e
const long double& e = d;

++d; // both d and e are changed: d is not constant, after all!
```

Summary

- All data is ultimately stored in **binary**, base 2, on a computer.
- In C++, there are many numeric types: the integer types, signed and **unsigned** version of **char**, **short**, **int**, **long int**, and **long long int**; the floating types of **float**, **double**, and **long double**; and the **bool type** and **char** types, and more....
- **size_t** variables are unsigned integer types that are the most appropriate choice whenever considering sizes or indices of containers.
- Through **static_cast**ing, we can convert an expression to another type.
- C++ offers a lot of basic arithmetic along with shorthand notation for updating a variable, incrementing, and decrementing, etc.
- **std::strings**, in the **<string>** header, can be used to manipulate strings of text and they include a lot of useful, built-in functions.
- Using **cin >>** can read in one entity or until there is a whitespace, and **getline** can collect an entire line of text.

Summary

- Output can be formatting using manipulators found in the **<iomanip>** header, and more advanced mathematical functions can be invoked from the **<cmath>** header.
- **Classes** are an important construct to group data and functions together and encapsulate said data.
- An **interface** of a class can be read to determine what a class is and how it works.
- A **header file** includes a listing and documentation for the different functions available.
- A **reference** gives another name for a variable.