

# C++ Data Types

## Contents

1	<i>Simple C++ Data Types</i>	1
2	Quick Note About Representations	2
3	Numeric Types	2
3.1	Integers (whole numbers)	3
3.2	Decimal Numbers	3
4	Text Types	5
4.1	Individual Character	5
4.2	Strings	5
5	Boolean Type	6
5.1	Note about Relational Operators	6
5.2	Boolean values as <i>numbers</i>	7
6	Casting: Interpreting one type as another	7
6.1	Automatic Casting vs Explicit Casting	7
7	Defining Constants	8

## 1 *Simple C++ Data Types*

- You've already seen and worked with a few data types
- Now we'll take a closer look at them
- We'll call these types *simple*, since (as you'll see later) C++ allows you to define your own data types
- Quick list of types C++ provides...

Name	C++ Keyword	Description	Example Values
Integers	<code>int</code>	Whole numbers, positive or negative	0, 42, -123
Floating Point Numbers	<code>float</code>	Decimal numbers	1.2, 0.0004, 123
Double Precision	<code>double</code>	<i>More precise</i> decimal numbers	1.2, 1234.5678
Characters	<code>char</code>	Individual textual characters	'a', 'B', '4'
Boolean	<code>bool</code>	Boolean values of true/false	true, false

- We'll also work with a C++ type `string`, which is *not a simple type!*
- To use this data type, you have to include

```
#include <string>
```

at the top of your program

- After that, the keyword is `string`, and example values are essentially lists of characters, such as `"hello world"`.

- As you've also seen, each data type has a set of *operators* that work with it.
  - These are things like addition (+), subtraction (-), etc.
  - There are also *relational operators* such as less-than (<), less-than-or-equal-to (<=), etc.
  - These are defined in the book (or the packet I'll hand out today), and I'll leave you to look over them
- 

## 2 Quick Note About Representations

- I'm not going to go into a great deal of depth about *how* data values such as 1, 2.5, 'a', true, and "hello world" are represented
  - The book details some of this, and you are required to read it
  - However, you should be aware of the following...
- 

- All data, regardless of the type, is represented in memory in *binary*
  - As such, if two sequences of binary bits in memory are the same, then clearly they have the same value... *but!*
  - They may represent different types of things!
  - As a quick example, the letter "A" in ASCII code (you don't need to know anymore about this than what the book describes) is ***represented by the number 65*** (in decimal).
  - As such, there's essentially no difference between the ***representation*** of those values
  - However, they are ***interpreted*** differently through the use of types.
  - *examples...*
  - That is, *types help give some intuitive interpretation of raw data*
- 

## 3 Numeric Types

- Numbers... they're everywhere.
  - Unfortunately, there are infinitely many of them
  - So for a finite machine to represent numbers in some *finite* fashion, clearly we can only represent finitely many
  - By having multiple types to express different sets of numbers, we can pick a representation than can express more such numbers
  - If that sounds confusing, we'll see more in a minute
-

### 3.1 Integers (whole numbers)

- Integers are whole numbers, either positive or negative.
  - Variables are declared integers using the C++ keyword `int`
  - **Expressive Power.** As mentioned above, we can only represent finitely many things...
  - There are two pre-defined constants that are available in Visual Studio (and GNU's `gcc/g++` suite) called `INT_MIN` and `INT_MAX`
  - *example...*
  - These represent the limits of `int`'s expressive powers
  - No numbers larger than `INT_MAX` or smaller than `INT_MIN` can be represented.
- 

- If you try to store too large or too small of a number in a variable of type `int`, you'll get what's called *overflow*
  - Depending upon the architecture and software, you'll either get total garbage as a result, or some "default value" such as 0
  - *examples...*
- 

### 3.2 Decimal Numbers

- Integers are nice...
  - But what about non-whole numbers, such as 0.5?
  - C++ provides two data types to represent such numbers, `float` and `double`
  - How are decimal numbers represented?
- 

- Answer: Scientific Notation
- Given any real number such as 1.2, we can represent this in scientific notation as

$$12 \times 10^{-1}$$

- Here, 12 is called the *coefficient* (also sometimes called the *mantissa*)
- The  $-1$  is called the *exponent*
- Notice that using this representation, we can represent any decimal number as a pair of two *whole numbers*
- Examples...

Number	Coefficient	Exponent
1.2	12	-1
0.005	5	-3
1002.3004	10023004	-4
100.0	1	2
100200.0	1002	2

- 
- So, *What's the difference between float and double?*
  - `double`'s use more memory and allocate more storage for both the mantissa and exponent
  - Since `double`'s can represent more, they require more memory
  - Also, working with `double`'s can be slower
  - *BUT!* they can express more
  - It's important to know when/where to use `float` vs. `double`
  - However for the purpose of this class, you can use `double` for virtually everything
- 

- **Expressive Power.** Unlike `int`'s, measuring the expressive power of these data types is not so easy
  - Since there are infinitely many decimal numbers between 0 and 1, we can't even represent all of those
  - As such, there's really no range like `FLOAT_MIN` to `FLOAT_MAX`
- 

- When using these data types, you can still have *overflow* just like we saw with `int`'s
- *examples...*
- In addition, we can also experience something called *underflow*...
- If you have a number  $n$  whose storage is already requiring most of the space allocated for a `float`/`double`, and you add to  $n$  some value  $m$  which is small (compared to  $n$ ), you may end up with the following...

`n == n + m`

- This is because C++ will try to keep only the *most significant* portion of the number
  - For example, if you have a number  $n$  such as 123,456,789 and you want to add to this number  $m$ , where  $m$  equals 0.0000000000004, the addition will essentially ignore  $m$
  - Notice, underflow may still cause rounding with least significant digits!
  - *examples...*
- 

- Question: What about infinitely-repeating decimals... such as  $\frac{1}{3}$ , or  $\pi$ ?
  - How do we represent them, if at all?
-

## 4 Text Types

- So, what about text?
  - Things like characters 'a', 'A', and strings such as "Hello World!"
  - C++ provides a data type called `char` to represent individual characters
  - To represent a *sequence* of characters (a.k.a. a "string"), you can do one of two things...
    1. Use the (non-simple) data type `string`
    2. Use C-style strings, which are essentially arrays of `char`'s
  - We're not going to cover the second method until later, when we cover arrays (even then we'll just skim the use of `char` arrays)
  - So for this class, just use `string`
- 

### 4.1 Individual Character

- C++ provides a data type called `char` to represent an individual such as 'a', 'B', or 'c'
  - Note that it is *case sensitive*
  - `char` can also represent some non-alpha characters, like '+', '!', '-', etc.
  - It can also represent numbers as *characters*
  - e.g. '1', '4'
  - However, `char` can *not* represent a sequence of characters!
  - So, `char` cannot store values like '42', 'so', etc.
- 

- To store a value as a `char`, notice that we put single quotes around the character
  - *examples...*
- 

### 4.2 Strings

- Strings are extremely common
- We frequently want to manipulate sequences of characters, and treat them like pieces of data
- To do this in C++, we'll be using the *Standard C++* library `string`
- To work with this, you must include the library in your files, as described above

```
#include <string>
```

- After that (and the usual `using namespace std;` declaration), you can use `string` just as you would any other data type
- 

- Keep in mind that `string` is *not* a simple type

- It's actually a *class*, which we'll talk briefly about throughout the course, and next quarter you'll learn more about those
- *examples...*

## 5 Boolean Type

- C++ provides the `bool` type which is used to represent the logical values of `true` and `false`
- Selection control structures such as `if` use boolean values to determine a course of action
- We'll use these conditional statements *heavily*

- Like with numbers, there are some common operations we do with boolean values which are no different than addition/subtraction for numbers
- These include the following...
  1. **NOT** Also known as *negation*, given a single boolean value, **not** returns the *opposite* value. In C++, the **not** operator is “!”
  2. **AND** Also known as a *conjunction*, given *two* boolean values, **and** return true if and only if *both of the input values are true*. Otherwise it return false. In C++ the **and** operator is “&&”.
  3. **OR** Also known as a *disjunction*, given two boolean values, **or** return true if and only if *at least one of the input values is true*. Otherwise it returns false. In C++ the **or** operator is “||”.
- The *truth-tables* for these operators are as follows. Given boolean variables `a` and `b`, the results of `!a`, `a && b`, and `a || b` are shown.

a	!a	a	b	a && b	a	b	a    b
true	false	true	true	true	true	true	true
true	false	true	false	false	true	false	true
false	true	false	true	false	false	true	true
false	true	false	false	false	false	false	false

### 5.1 Note about Relational Operators

- From what we've seen so far, the arithmetic operators like addition (+), subtraction (-), etc., all take inputs and return values.
- i.e. they are functions
- That is, given the numbers 5 and 6, `5 + 6` returns 11.
- The relational operators (<, <=, ==, etc.) are no different
- Except that instead of returning numbers, they return *boolean* values
- *examples...*

## 5.2 Boolean values as *numbers*

- In C, there is no boolean data type
  - Since C++ is a superset of C, it also incorporates this “feature”
  - Instead of using such a data type in C, `int`'s are used
  - Integer expressions are interpreted as boolean values as follows
    1. Zero is false
    2. *Anything else is true*
  - *examples...*
  - That's it...you should know about this (the book has a nice one page description of it), but that's about it
- 

## 6 Casting: Interpreting one type as another

- Sometimes you'll have a variable of one type, but want to *cast* the value to another type.
  - For example, you may want to case a `float` to an `int`
  - As we saw yesterday, this is perfectly legal...
  - But be aware that there may be some loss of information
  - Since `float`'s can represent decimal values, but `int`'s can only represent whole values, when you perform such a cast the decimal part is completely ignored
- 

### 6.1 Automatic Casting vs Explicit Casting

- In many cases, C++ can do an automatic cast between all of it's *simple* data types
  - That is, if you try to coerce one type of number (say a `float`) to another (say an `int`), C++ will do that just fine
  - This is called *automatic casting*
  - However, because of the possible loss of information discussed above, C++ will kindly warn you that you may be losing information
  - To remove the warnings, you can use an *explicit cast*, which basically informs the C++ compiler that you know what's happening and expect the consequences
- 

- For example, suppose you had a `float` or `double` that you *knew was a whole value...*and you'd like to turn the value into an `int`
- You could do the following, making use of *automatic casting...*

```
double d = 123;  
int i;  
  
i = d;
```

- That’s a valid C++ statement
- **BUT!** C++ will generate a warning saying (essentially) that you have a “type casting with possible loss of data”
- ... Since C++ doesn’t know what you know (that `d` has a whole-number value), it alerts you that something unintended may be happening
- To avoid this, use an *explicit cast*, such as follows

```
double d = 123;
int i;

i = (int) d;
```

- The warning will go away, as you are now explicitly telling C++ that a cast should happen
- *examples...*
- There are more types of casting described in the book, such as `static_cast< type >`, and you should be aware of these... *they may pop up on a quiz*

## 7 Defining Constants

- We’ve seen some of the simple data types that C++ provides, and even a non-simple data type (strings)
- Now we look at a *type modifier* that C++ provides
- There are other type modifiers provided by C++ that the book describes, but most are not introduced until later
- For now, you just need to know this one

- When we write pieces of software, there may be occasions when we’d like to give a variable a specific value, and make it *constant*
- That is, there may be values we’ll use that we *never want to change*
- For example, if we’re writing a program to calculate the area or circumference of a circle after a user inputs a radius, we’d like to make use of  $\pi$
- So we might store  $\pi$  (well, some part of  $\pi$ ) in a variable
- *And we wouldn’t want that to change*

- Now, we could adopt the policy that “nobody ever changes the variable named `pi!!!!`” as a life-or-death rule
- But with large pieces of software, the code may be broken up among *many files*, being worked on by *many people*
- C++ helps by providing a modifier to make a value *constant*

- Once the variable is set, *it can never change in your program*
- 

- The type modifier we're looking at is `const`, and to declare constant variables we do the following...

```
const SomeType SomeVariableName = SomeValue;
```

Examples...

```
const int myStreetAddress = 1234;  
const string myStreetName = "Sesame Street";  
const long double PI = 3.1415926535897932384626433832795;  
const double interestRate = 0.88; // Ouch!
```

- So the `const` goes *before* the type, and the variable *must be set when it's declared!*
- **NOTE:** Failure to set a constant variable in its declaration will result in an unassigned variable *that can never change!*
- *examples...*