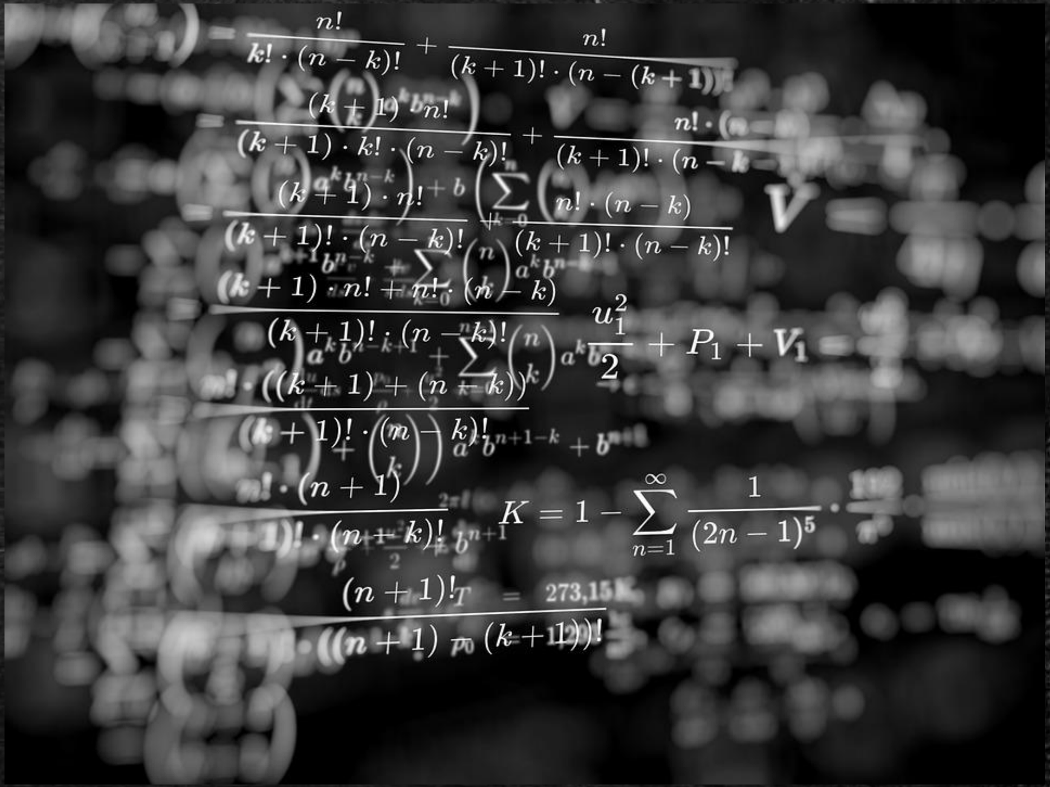
Numbers & Calculations



$\frac{n!}{k! \cdot (n-k)!} + \frac{n!}{(k+1)! \cdot (n-(k+1))!}$
$\frac{(k+1) \cdot n!}{(k+1) \cdot k! \cdot (n-k)!} + \frac{n! \cdot (n-k)}{(k+1)! \cdot (n-k)!}$
$\frac{(k+1) \cdot n!}{(k+1)! \cdot (n-k)!} + \frac{n! \cdot (n-k)}{(k+1)! \cdot (n-k)!}$
$\frac{(k+1) \cdot n! + n! \cdot (n-k)}{(k+1)! \cdot (n-k)!}$
$\frac{u_1^2}{2} + P_1 + V_1$
$\frac{n! \cdot ((k+1) + (n-k))}{(k+1)! \cdot (n-k)!}$
$\frac{n! \cdot (n+1)}{(k+1)! \cdot (n+1-k)!}$
$K = 1 - \sum_{n=1}^{\infty} \frac{1}{(2n-1)^5}$
$\frac{(n+1)!}{((n+1)-(k+1))!}$
$T = 273{,}15\,K$

CS 150 – C++ Programming I
Lecture 3

Type Concepts

- Types: a) domain, b) operations, c) representation
  - a) domain – all values contained in type
  - b) operations (specified by C++ language for built-in)
  - c) representation (and size) differs by implementation
- Categories of types in C++
  - Built-in, primitive or fundamental types
    - integers, floating-point, `char`, `bool`
  - Derived, compound types: array, pointer, reference
  - User-defined types: `enum`, `struct`, `class`
  - Library types such as `string` & `vector`

Static, Dynamic & Strong Typing

- C++ is statically typed: types indicated in source
  - Dynamic typing: type determined at runtime (Python)
    - `def add(a, b): // types calculated when run`
  - Static typing: type explicitly specified in declaration
    - `int add(int a, int b);  // types when compiled`
- C++ variables are strongly typed with implicit conversions
  - `int a = 3.5; // OK, narrowing conversion`
  - `int b = "Hello World"; // No conversion`

The Built-in Numeric Types

- `int` is a signed whole number
  - Must be >= 16 bits (32 most common)
  - `long` (>= 32), `short` (>= 16), `long long` (>= 64)
  - `unsigned` (combine with those above)
  - Byte integers, `signed` or `unsigned char`
- C++ *floating-point* types:
  - `float` (>=4), `double` (typically 8), `long double`

Literals

- Literals assume working with decimal numbers (base 10)
  - Prefix modifiers: `073` (octal), `0x73` (hex), `0b111` (binary)
  - Suffix modifiers: `123U`, `123L`, `123LL`, `123ULL`,
    - C++ 14 also allows `'` to act as a separator for large numbers
- Floating-point literals
  - `234.`: type `double`, no trailing `0` required
  - `7.5432L`: type `long double`
  - `1.234e-12`: `double` using scientific notation
  - `0.25F`: type `float`

Inferred Typing

- Starting with C++11 you can infer or deduce the type
  - Use the keyword auto as the type and don't use braces
  - `auto a = 23U;          // a is unsigned int`
  - `auto b = 3.5F;     // b is float`
- Many modern C++ experts recommend AAA because:
  - Correctness: eliminates uninitialized variables
  - Maintenance: types "track" as initializers are changed
  - Performance: eliminates unnecessary implicit conversions
  - Read more here, here and here, or watch this
- Exercise: deduce types from literals

Binary Numbers

- Internally, all numbers are stored in binary (base 2)
  - Base 2: 0,1; 8: 0-7; 10: 0-9; 16: 0-F
- Simplest integer representation is unsigned
  - Base 10 representation: 123
    - $(1 * 10^2) + (2 * 10^1) + (3 * 10^0)$
  - Base 2 (binary) representation: 1111011
    - $2^6 + 2^5 + 2^4 + 2^3 + 0 + 2^1 + 2^0$
    - 64 + 32 + 16 + 8 + 2 + 1 -> 123
  - We'll normally use base 10, but you should know algorithms to convert to different bases
- Exercise: examine some representations

Conversions and Casts

- C++ automatically converts between types of numbers on assignment or initialization
  - This implicit conversion happens without warning
  - In C++11, use list assignment for greater control
  - May get compiler warning on narrowing conversion

- Explicitly convert using `static_cast`
  - Shows that conversion is intentional

```
double pi = 3.14159;
 int x = static_cast<int>(pi)
```

Expressions

- Operators and operands are evaluated to produce new values
  - Operands may be:
    - Literals `3 + 5` [3 and 5 are literal values]
    - Variables `a - 3` [a contains a value]
    - Function call `2 * func(a)` [func() produces a value]
    - Subexpression `(a + 3) * 5` [a + 3 results in a value]
  - Operators have three characteristics
    - arity: how many operands are needed (unary, binary, tertiary)
    - precedence: which bind more tightly to data (PMA)
    - associativity: left-to-right or right-to-left
- Example: what is `7 * 2  / 3`?

Basic Arithmetic Operators

- There are five binary arithmetic operators
  - Don't modify operands so expressions and literals OK
  - addition (+), subtraction (-). (Also used as unary operators)
  - multiplication (*), division (/) and remainder (%)
- Integer division resembles primary school long-division
  - The quotient is calculated, remainder discarded
  - The result, an integer, is truncated, not rounded
- The % operator returns the remainder part
  - 12 % 5 is 2 because there is 2 left over after dividing 12 by 5

Increment and Decrement

- Unary operators: ++ and - -
  - Primary purpose: produce a value
  - Secondary (side effect): change operand
    - Thus these only work with modifiable lvalues
    - Adds or subtracts one to the variable's value
- Value of the expression depends on operator placement
  - Prefix: (++a) change variable then return changed variable
  - Postfix or suffix: (a++) save old value as temporary, change the variable, use the saved temporary
  - Result of postfix is not an lvalue (but prefix is!)

Increment/Decrement Pitfalls

- Things to avoid:
  - `printThis(n, n * n++);`
  - `ans = n / 2 + 5 * (1 + n++);`
  - `y = n++ + n++;`
- Some rules to remember:
  - Don't use on a variable that is part of more than one argument
  - Don't use if a variable appears more than once in an expression
  - Also applies to the assignment operator
  - `y = y++; // what does this even mean?`
- Exercise: Calculating Calories