Introduction to C

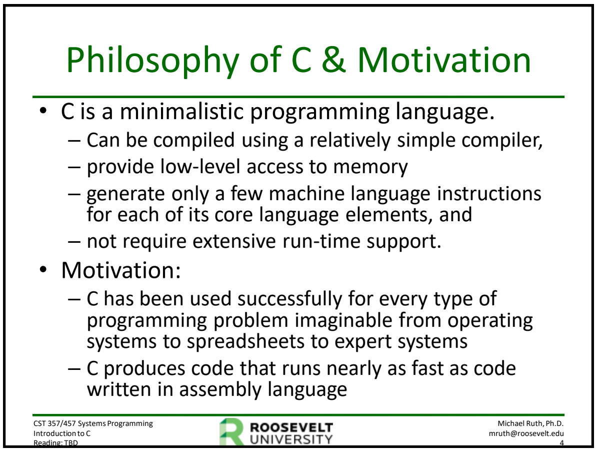
CST 357/457 – Systems Programming
Michael Ruth, Ph.D.
Associate Professor
Computer Science & I.T.
mruth@roosevelt.edu

Objectives

- Introduce the programming language C and discuss its advantages/disadvantages
- Explain variables and data types in C including declarations and operations
- Discuss control flow concepts in C including selection and iteration statements
- Explain the use of functions in C including declaration, prototypes, and recursion
- Discuss scope and variable initialization

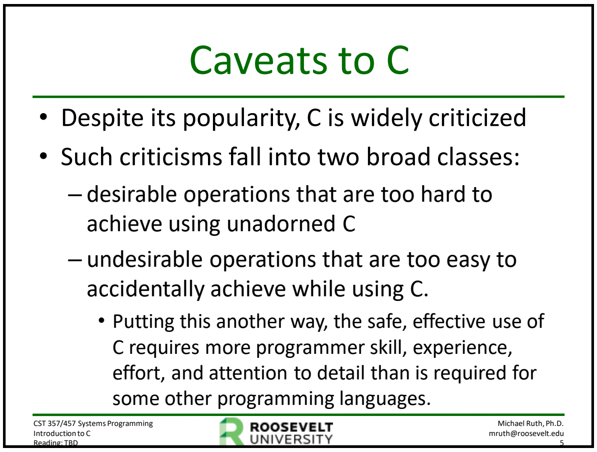
Introduction to C Programming Language

- C is a general-purpose, procedural, imperative computer programming language
  - Procedural is a paradigm based on the notion of function calls
    - Call functions to get your work done
  - Imperative is a paradigm that describes computation as statements that change a program state
- Notice that there is NO OO here?
  - C++ added objects among other things to C



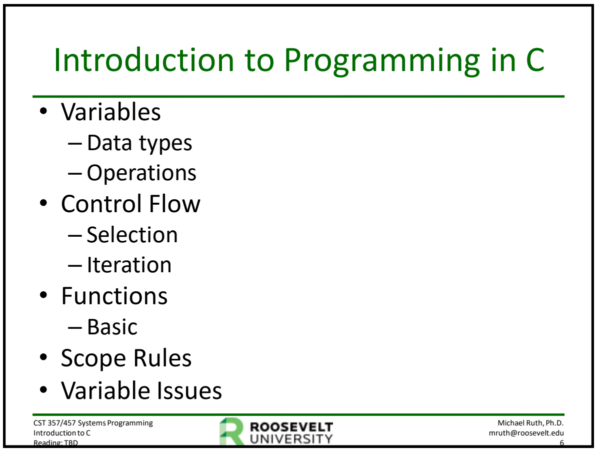
Philosophy of C & Motivation

- C is a minimalistic programming language.
  - Can be compiled using a relatively simple compiler,
  - provide low-level access to memory
  - generate only a few machine language instructions for each of its core language elements, and
  - not require extensive run-time support.
- Motivation:
  - C has been used successfully for every type of programming problem imaginable from operating systems to spreadsheets to expert systems
  - C produces code that runs nearly as fast as code written in assembly language



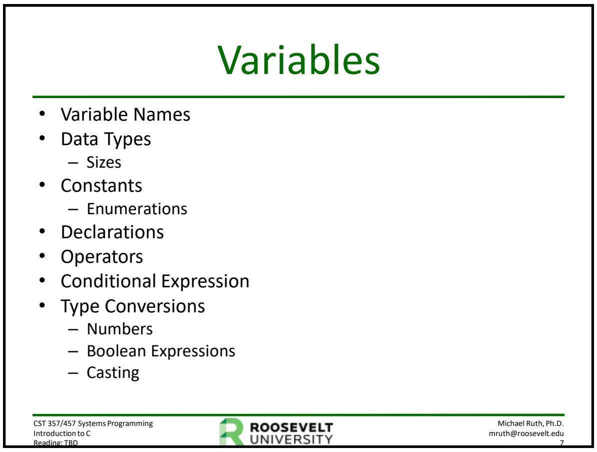
Caveats to C

- Despite its popularity, C is widely criticized
- Such criticisms fall into two broad classes:
  - desirable operations that are too hard to achieve using unadorned C
  - undesirable operations that are too easy to accidentally achieve while using C.
    - Putting this another way, the safe, effective use of C requires more programmer skill, experience, effort, and attention to detail than is required for some other programming languages.



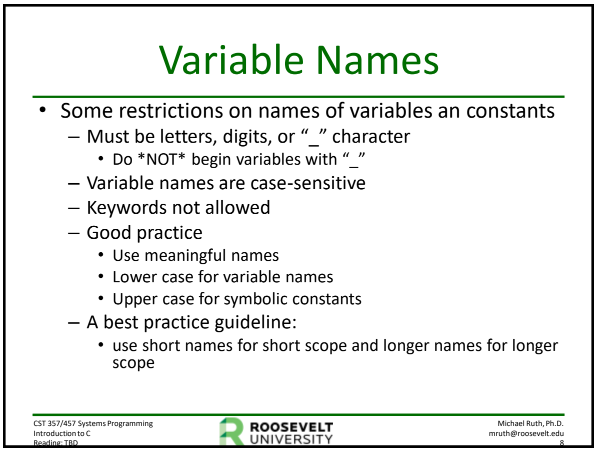
Introduction to Programming in C

- Variables
  - Data types
  - Operations
- Control Flow
  - Selection
  - Iteration
- Functions
  - Basic
- Scope Rules
- Variable Issues



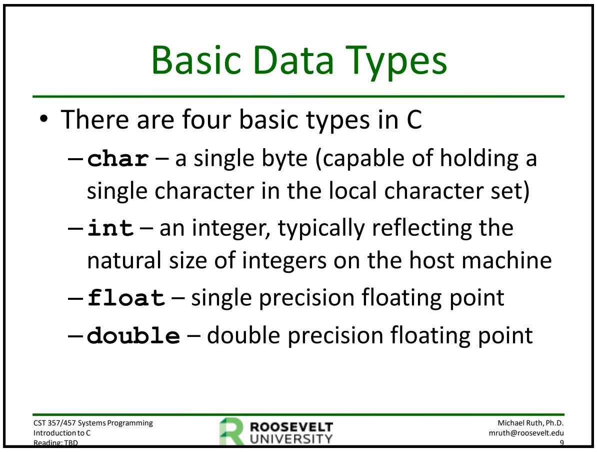
Variables

- Variable Names
- Data Types
  - Sizes
- Constants
  - Enumerations
- Declarations
- Operators
- Conditional Expression
- Type Conversions
  - Numbers
  - Boolean Expressions
  - Casting



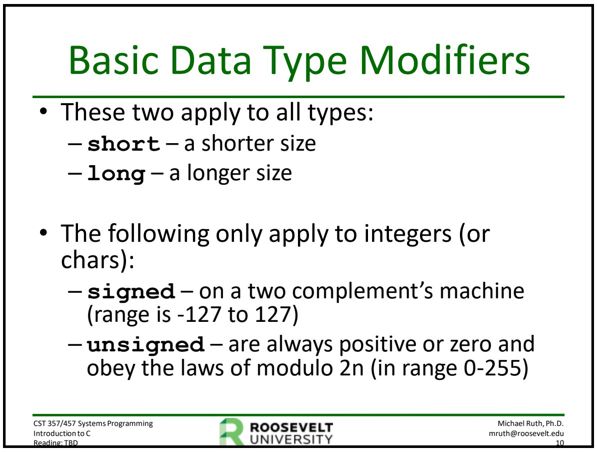
Variable Names

- Some restrictions on names of variables an constants
  - Must be letters, digits, or "_" character
    - Do *NOT* begin variables with "_"
  - Variable names are case-sensitive
  - Keywords not allowed
  - Good practice
    - Use meaningful names
    - Lower case for variable names
    - Upper case for symbolic constants
  - A best practice guideline:
    - use short names for short scope and longer names for longer scope



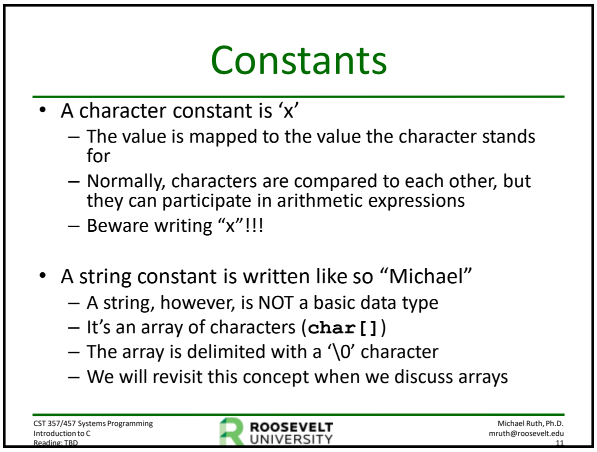
Basic Data Types

- There are four basic types in C
  - **`char`** – a single byte (capable of holding a single character in the local character set)
  - **`int`** – an integer, typically reflecting the natural size of integers on the host machine
  - **`float`** – single precision floating point
  - **`double`** – double precision floating point



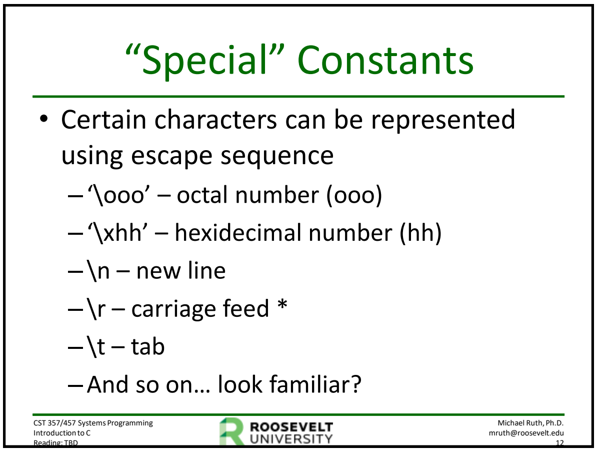
Basic Data Type Modifiers

- These two apply to all types:
  - **short** – a shorter size
  - **long** – a longer size
- The following only apply to integers (or chars):
  - **signed** – on a two complement's machine (range is -127 to 127)
  - **unsigned** – are always positive or zero and obey the laws of modulo 2n (in range 0-255)



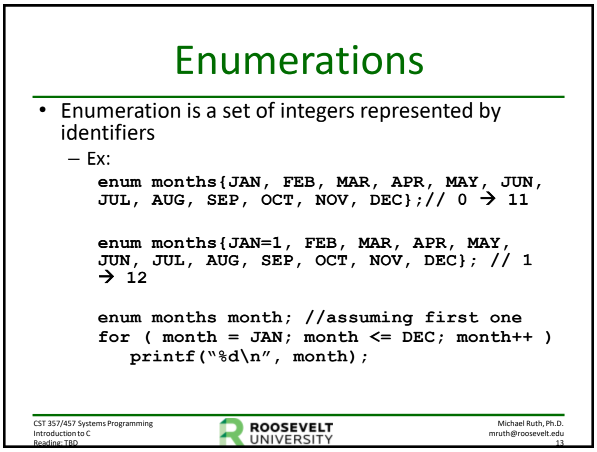
Constants

- A character constant is 'x'
  - The value is mapped to the value the character stands for
  - Normally, characters are compared to each other, but they can participate in arithmetic expressions
  - Beware writing "x"!!!
- A string constant is written like so "Michael"
  - A string, however, is NOT a basic data type
  - It's an array of characters (**char[]**)
  - The array is delimited with a '\0' character
  - We will revisit this concept when we discuss arrays



"Special" Constants

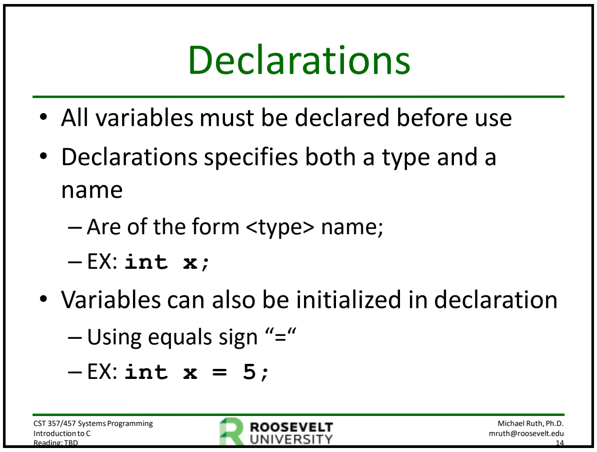
- Certain characters can be represented using escape sequence
  - '\ooo' – octal number (ooo)
  - '\xhh' – hexidecimal number (hh)
  - \n – new line
  - \r – carriage feed *
  - \t – tab
  - And so on… look familiar?



Enumerations

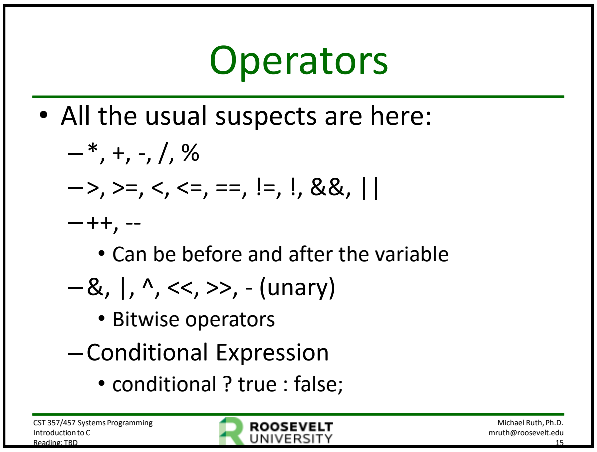
- Enumeration is a set of integers represented by identifiers
  - Ex:

```
enum months{JAN, FEB, MAR, APR, MAY, JUN,
JUL, AUG, SEP, OCT, NOV, DEC};// 0 → 11

enum months{JAN=1, FEB, MAR, APR, MAY,
JUN, JUL, AUG, SEP, OCT, NOV, DEC}; // 1
→ 12

enum months month; //assuming first one
for ( month = JAN; month <= DEC; month++ )
   printf(“%d\n”, month);
```



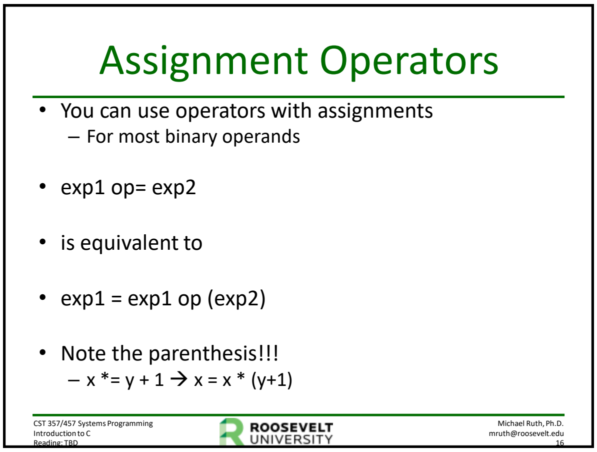
Declarations

- All variables must be declared before use
- Declarations specifies both a type and a name
  - Are of the form <type> name;
  - EX: **`int x;`**
- Variables can also be initialized in declaration
  - Using equals sign “=“
  - EX: **`int x = 5;`**



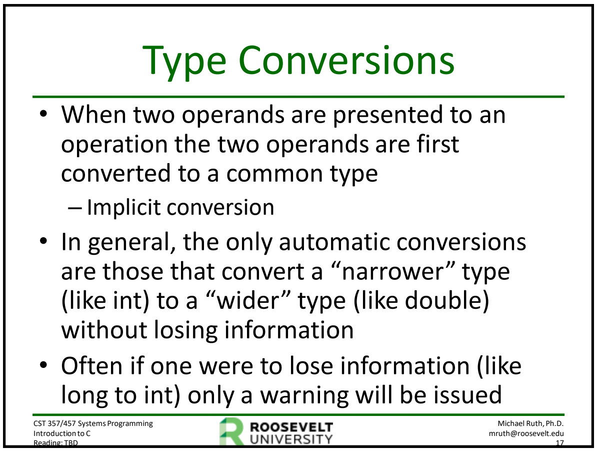
Operators

- All the usual suspects are here:
  - *, +, -, /, %
  - >, >=, <, <=, ==, !=, !, &&, ||
  - ++, --
    - Can be before and after the variable
  - &, |, ^, <<, >>, - (unary)
    - Bitwise operators
  - Conditional Expression
    - conditional ? true : false;



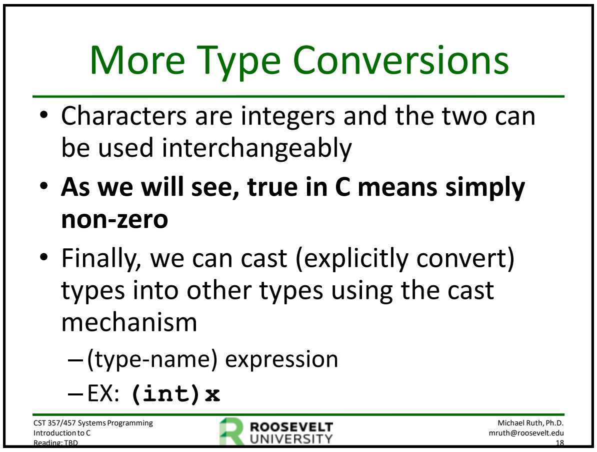
Assignment Operators

- You can use operators with assignments
  - For most binary operands
- exp1 op= exp2
- is equivalent to
- exp1 = exp1 op (exp2)
- Note the parenthesis!!!
  - x *= y + 1 → x = x * (y+1)



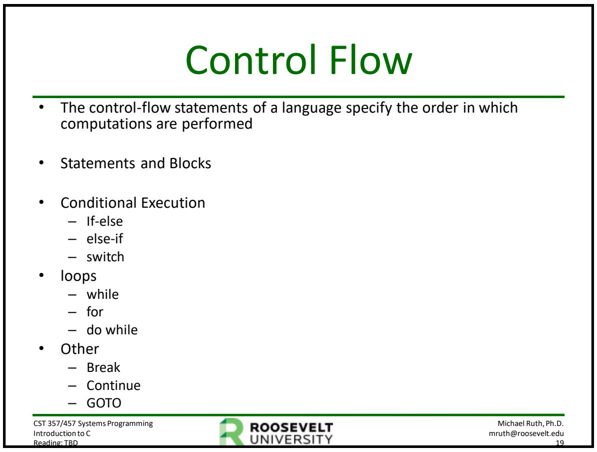
Type Conversions

- When two operands are presented to an operation the two operands are first converted to a common type
  - Implicit conversion
- In general, the only automatic conversions are those that convert a “narrower” type (like int) to a “wider” type (like double) without losing information
- Often if one were to lose information (like long to int) only a warning will be issued



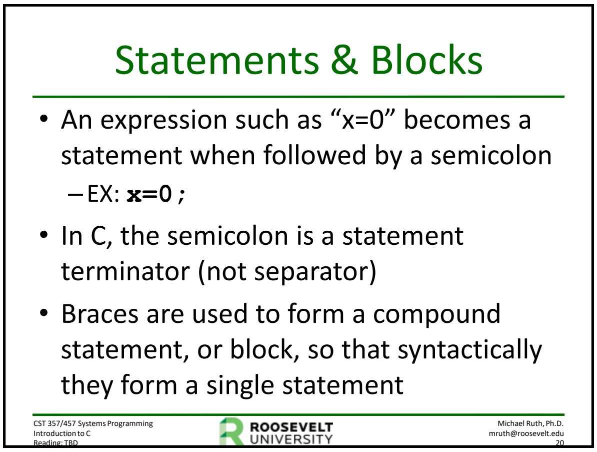
More Type Conversions

- Characters are integers and the two can be used interchangeably
- **As we will see, true in C means simply non-zero**
- Finally, we can cast (explicitly convert) types into other types using the cast mechanism
  - (type-name) expression
  - EX: `(int)x`



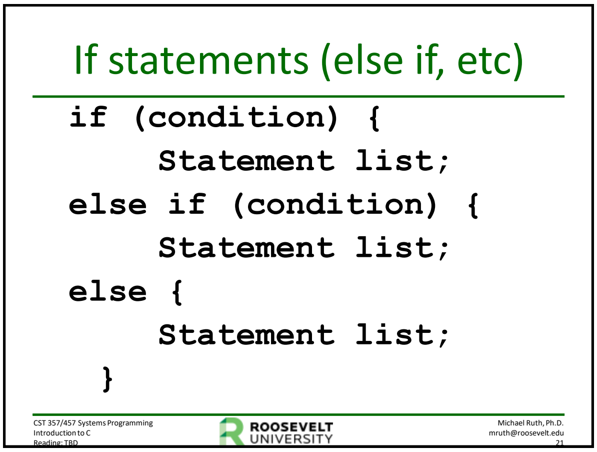
Control Flow

- The control-flow statements of a language specify the order in which computations are performed
- Statements and Blocks
- Conditional Execution
  - If-else
  - else-if
  - switch
- loops
  - while
  - for
  - do while
- Other
  - Break
  - Continue
  - GOTO



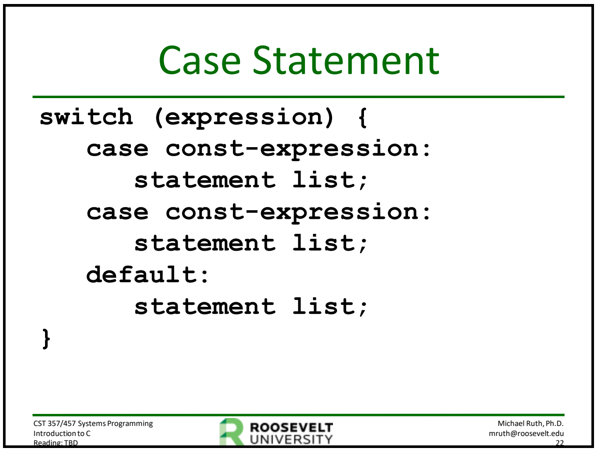
Statements & Blocks

- An expression such as “x=0” becomes a statement when followed by a semicolon
  - EX: **`x=0;`**
- In C, the semicolon is a statement terminator (not separator)
- Braces are used to form a compound statement, or block, so that syntactically they form a single statement



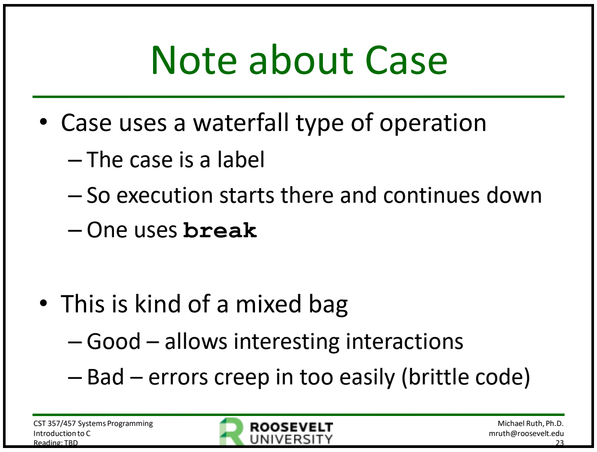
If statements (else if, etc)

```
if (condition) {
     Statement list;
else if (condition) {
     Statement list;
else {
     Statement list;
  }
```



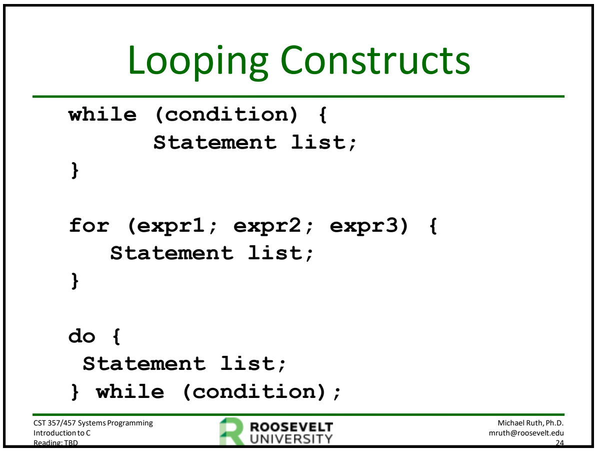
Case Statement

```
switch (expression) {
   case const-expression:
      statement list;
   case const-expression:
      statement list;
   default:
      statement list;
}
```



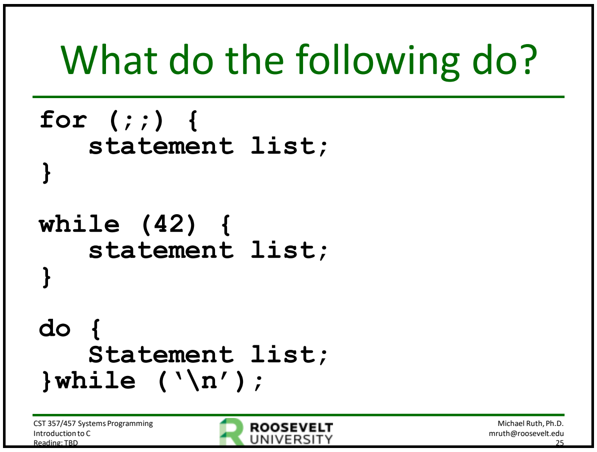
Note about Case

- Case uses a waterfall type of operation
  - The case is a label
  - So execution starts there and continues down
  - One uses **`break`**
- This is kind of a mixed bag
  - Good – allows interesting interactions
  - Bad – errors creep in too easily (brittle code)



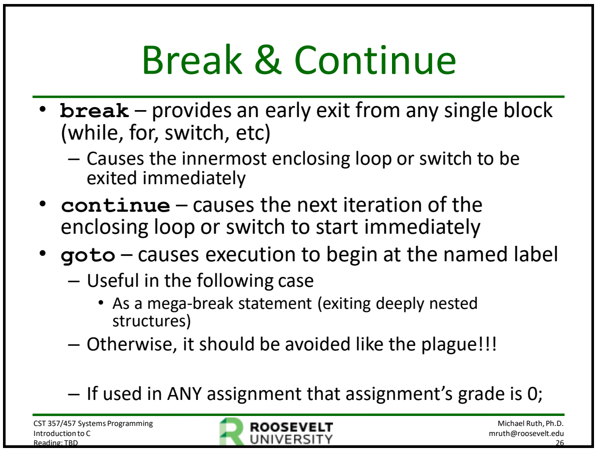
Looping Constructs

```
while (condition) {
      Statement list;
}

for (expr1; expr2; expr3) {
   Statement list;
}

do {
 Statement list;
} while (condition);
```



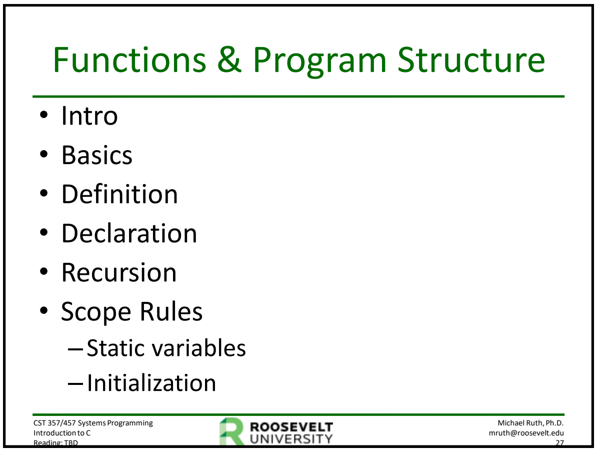
What do the following do?

```
for (;;) {
   statement list;
}

while (42) {
   statement list;
}

do {
   Statement list;
}while (‘\n’);
```



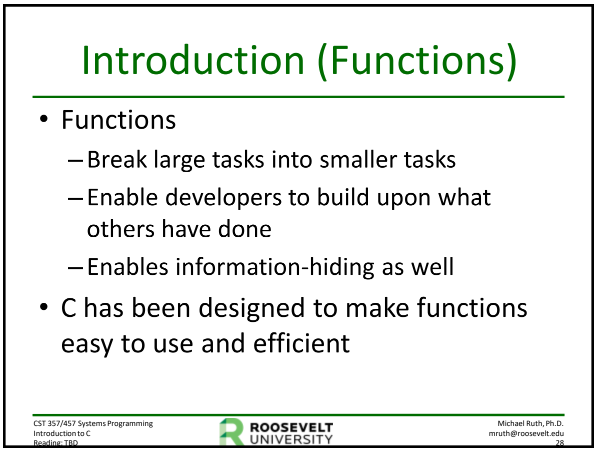
Break & Continue

- **break** – provides an early exit from any single block (while, for, switch, etc)
  - Causes the innermost enclosing loop or switch to be exited immediately
- **continue** – causes the next iteration of the enclosing loop or switch to start immediately
- **goto** – causes execution to begin at the named label
  - Useful in the following case
    - As a mega-break statement (exiting deeply nested structures)
  - Otherwise, it should be avoided like the plague!!!
  - If used in ANY assignment that assignment's grade is 0;



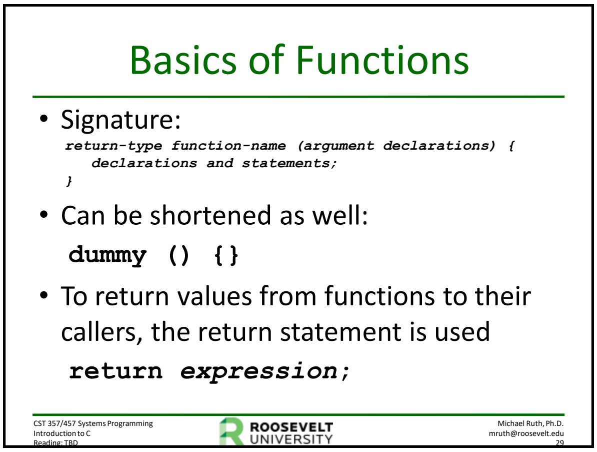
Functions & Program Structure

- Intro
- Basics
- Definition
- Declaration
- Recursion
- Scope Rules
  - Static variables
  - Initialization



Introduction (Functions)

- Functions
  - Break large tasks into smaller tasks
  - Enable developers to build upon what others have done
  - Enables information-hiding as well
- C has been designed to make functions easy to use and efficient



Basics of Functions

- Signature:

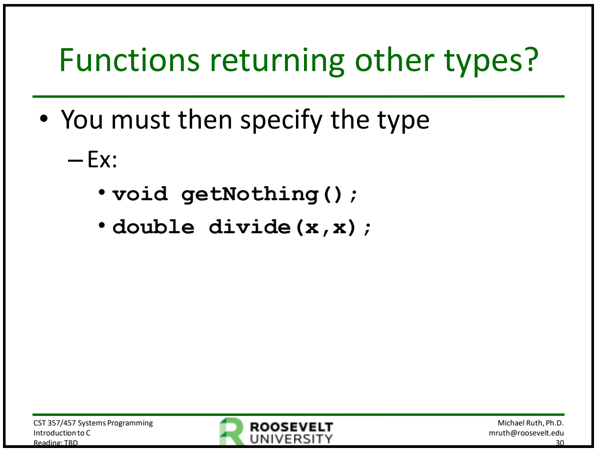
```
return-type function-name (argument declarations) {
    declarations and statements;
}
```

- Can be shortened as well:

```
dummy () {}
```

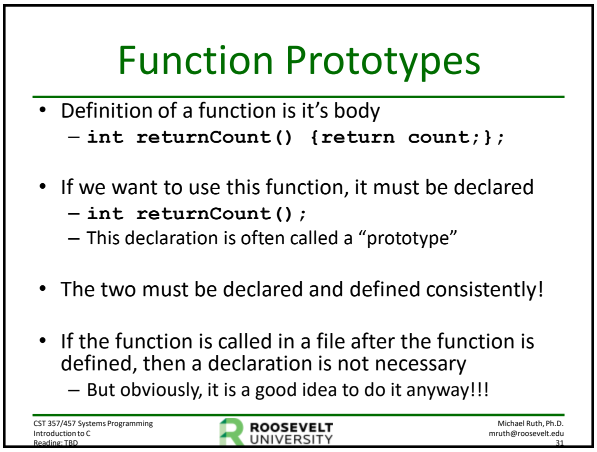
- To return values from functions to their callers, the return statement is used

```
return expression;
```



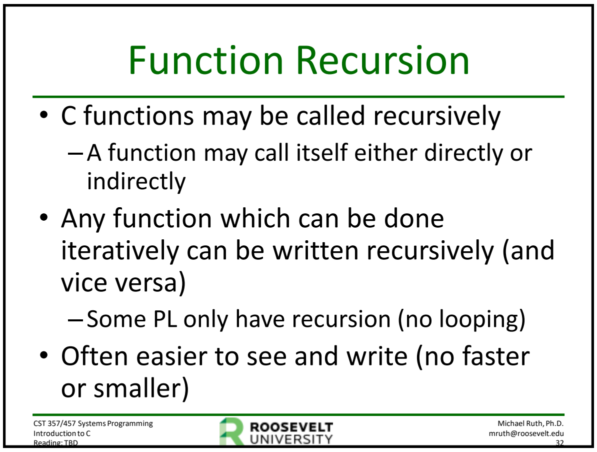
Functions returning other types?

- You must then specify the type
  - Ex:
    - `void getNothing();`
    - `double divide(x,x);`



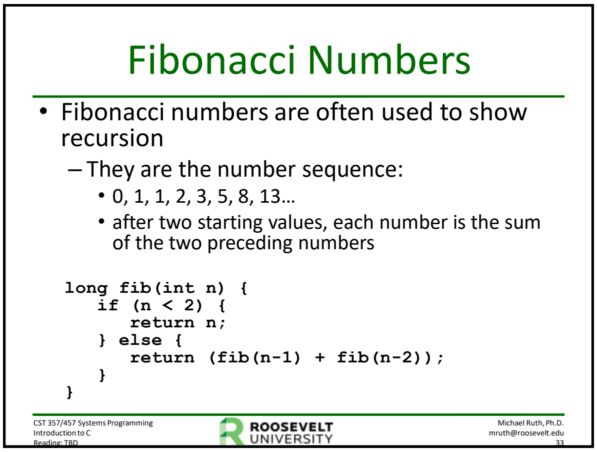
Function Prototypes

- Definition of a function is it's body
  - `int returnCount() {return count;};`
- If we want to use this function, it must be declared
  - `int returnCount();`
  - This declaration is often called a "prototype"
- The two must be declared and defined consistently!
- If the function is called in a file after the function is defined, then a declaration is not necessary
  - But obviously, it is a good idea to do it anyway!!!



Function Recursion

- C functions may be called recursively
  - A function may call itself either directly or indirectly
- Any function which can be done iteratively can be written recursively (and vice versa)
  - Some PL only have recursion (no looping)
- Often easier to see and write (no faster or smaller)



Fibonacci Numbers

- Fibonacci numbers are often used to show recursion
  - They are the number sequence:
    - 0, 1, 1, 2, 3, 5, 8, 13…
    - after two starting values, each number is the sum of the two preceding numbers

```
long fib(int n) {
   if (n < 2) {
      return n;
   } else {
      return (fib(n-1) + fib(n-2));
   }
}
```

Scoping Rules

- Now that we've discussed functions, we can comfortably discuss scope
  - – And variable modifiers to alter scoping rules
- The scope of a name is the part of the program within which that name may be used
- In general, the variable in question is only valid within the same block in which it is defined/declared
  - – If the variable is inside a block, function, it an automatic variable
  - – If the variable is outside this it is termed an external variable
- Additionally, the variable exists inside its area from the point it is declared until the end of the area it exists in

Static & Register variables

- Static modifier can be applied to both external and automatic names
  - – Static applied to an external variable limits the scope to the file in which it is compiled
    - Can also be applied to functions
  - – Static applied to an automatic variable allows the variable to remain in existence rather than being recreated every time the function is called
- Register modifier ***advises*** the compiler that the variable in question will be heavily used

Initialization

- We have discussed it but only slightly
- In the absence of explicit initialization, external and static variables are GUARANTEED to be zero
  - – Automatic and register have GARBAGE values
  - – For external and static variables, the initializer must be a constant expression

Summary

- Introduced the programming language C and discuss its advantages/disadvantages
- Explained variables and data types in C including declarations and operations
- Discussed control flow concepts in C including selection and iteration statements
- Explained the use of functions in C including declaration, prototypes, and recursion
- Discussed scope and variable initialization

Questions?