

Introduction to C Programming

Mitchell Trope

Licensing

- Copyright © 2009 Mitchell Trope
- This work is licensed under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/us/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Getting Started With C

- What is C and why should I use it?
 - C is one of the most popular programming languages in the world
 - A C compiler is available for almost any computing platform you can think of
 - It's suitable for low-level programming tasks like operating systems (Windows) or high-level applications (like PowerPoint)
 - C is especially useful for embedded systems, such as robots, because it creates very efficient code

How Do Computers Work?

- Computers have two fundamental states
 - On – 1, true, high
 - Off – 0, false, low
- The basic system is called “binary”
 - “Binary” is derived from the Latin term *binarius*
- Everything computers do can be described as collections of ones and zeros
- Each one or zero is called a “bit”
- Bits can be grouped in many different ways to represent data

Primary Computer Systems

- CPU – Central Processing Unit
 - Performs all the calculations your programs tell it to do
- Main memory
 - Storage for the program code that is running
 - Storage for the data all the programs need
 - Expensive per byte but pretty fast
- Bulk storage
 - Can store programs that aren't actively running or data that is not immediately needed
 - Hard drives, USB jump drives, SD cards
 - Cheap per byte but slow

How Do Computers Run Programs?

- The CPU can do several tasks natively
 - Math: add, subtract, multiply, divide
 - Comparisons: greater than, less than, equal to
 - Load/store data from memory
- A CPU's basic language is “assembly code”
 - Assembly code must be converted to “object code” for the CPU to execute it
- A program is a collection of these assembly instructions
 - Assembly instructions all tell the CPU what to do and which data to use
 - Assembly code is hard to use, and is only written by hand in special cases
 - We use languages like C to write the majority of our code

How Does C Become Something The CPU Runs?

- The C language is *compiled* in to machine code
- The C “compiler” is actually four steps
 - Pre-processor – handle *include files* and *constants* that give C its portability
 - Compiler – change *general C code* (which could be for any type of processor) in to *assembly code* (which is only for one type of processor)
 - Assembler – change the *assembly code* created by the compiler in to *object code* that the computer can understand
 - Linker – take all the *object code* and put it together with *libraries* to form your program

C Language Rules

- C has a grammar and syntax that you must follow so the computer understands what you want it to do
 - You have to follow the grammar and syntax of English so others can understand you when you communicate
- The compiler interprets that grammar and syntax
- The other stages of compilation also have their own rules
 - The pre-processor and assembler have their own syntax rules
 - The linker may not be able to find all your code if you haven't told it where to look
- Even if you use valid syntax, your program may still not function correctly
 - The computer will always follow your instructions, even if your instructions aren't what you intended

Example Program – Source Code

```
/* Program to calculate the product of two numbers */
#include <stdio.h>

int main()
{
    int a, b, c;

    /* Input the first number */
    printf( "Enter a number between 1 and 100: " );
    scanf( "%d", &a );

    /* Input the second number */
    printf( "Enter another number between 1 and 100: " );
    scanf( "%d", &b );

    /* Calculate and display product */
    c = product( a, b );
    printf( "%d times %d = %d\n", a, b, c );

    return 0;
}

/* Function returns the product of its two arguments */
int product( int x, int y )
{
    return( x * y );
}
```

What does
all this do?

Example Program

```
/* Program to calculate the product of two numbers */
#include <stdio.h>

int main()
{
    int a, b, c;

    /* Input the first number */
    printf( "Enter a number between 1 and 100: " );
    scanf( "%d", &a );

    /* Input the second number */
    printf( "Enter another number between 1 and 100: " );
    scanf( "%d", &b );

    /* Calculate and display product */
    c = product( a, b );
    printf( "%d times %d = %d\n", a, b, c );

    return 0;
}

/* Function returns the product of its two arguments */
int product( int x, int y )
{
    return( x * y );
}
```

Statements –
What the
program
actually does

Function – A
group of
statements
you can call
over and over

Comments – remarks
to yourself and others
so you remember what
you wanted to do or
how something should
work

Function prototype –
how should I ask the
program to run this
task?

Example Program

Include file –
pull in a group
of existing
stuff we want
to use

```
/* Program to calculate the product of two numbers */  
#include <stdio.h>
```

```
int main()  
{
```

```
    int a, b, c;
```

```
    /* Input the first number */
```

```
    printf( "Enter a number between 1 and 100: " );
```

```
    scanf( "%d", &a );
```

```
    /* Input the second number */
```

```
    printf( "Enter another number between 1 and 100: " );
```

```
    scanf( "%d", &b );
```

```
    /* Calculate and display product */
```

```
    c = product( a, b );
```

```
    printf( "%d times %d = %d\n", a, b, c );
```

```
    return 0;
```

```
}
```

```
/* Function returns the product of its two arguments */
```

```
int product( int x, int y )
```

```
{
```

```
    return( x * y );
```

```
}
```

Variables – data we
want to work with

Braces –
define the
scope of a
function

Programming Basics

- Variables
 - Variables are data we want to manipulate
 - The value can *vary* over the course of the program's execution
 - Names should be lower-case
- Constants
 - Constants are pieces of data we need to have but don't need to change
 - Names should be upper case so you know it's a constant, not a variable

Programming Basics - Variables

- Declaring a variable or group of variables

- `int a;`

- This line declares one variable named `a` of type *int*, or integer

- `int a, b, c;`

- This line declares three variables named `a`, `b` and `c`, of type *int*, or integer

Programming Basics - Variables

- *int* is one of several built in, or primitive, types
- Declarations are of the form <type> <name>; or <type> <name_1>, <name_2>, ..., <name_x>;
- Other primitive types
 - *char* – a one byte (8 bit) character, sign bit is most significant
 - *unsigned char* – one byte, no sign bit
 - *int* – signed integer, sign bit is most significant, size varies by platform
 - *unsigned int* – unsigned integer, no sign bit, size varies by platform
 - *long* – a long integer, size varies by platform
 - *float* – floating point number, used to represent values like 32.05

Programming Basics – Primitive Types

- Different data types are used based on what type of information we want to represent
- Data types are also selected for efficiency
 - Processors are most efficient in their native data size
 - Most processors today use 32 bit data words
 - Floating point data is much harder to process than integer data
 - An operation with floating point data can take several times the number of cycles as the same operation with integer data
 - Don't use floats when integers will do!

Programming Basics – Primitive Types

- The data type determines the amount of memory your variable is given and how the information is interpreted by the program
- Different types have different ranges and resolutions

32 Bit Signed Integer

- Most significant bit is the sign bit
 - 0 is positive, 1 is negative
- The other 31 bits are the remaining data
 - The range is -2147483648 to 2147483647
 - Why is the positive range smaller?

- 0110 1001 0010 0100 1100 1001 1110 0100
1,764,018,660 in decimal
 - 1110 1001 0010 0100 1100 1001 1110 0100
-383,464,984 in decimal
-

Floating Point Single Precision

- Three components
 - Sign bit – positive (0) or negative (1)
 - Exponent – signed, 8 bits -126 to 127
 - Fraction – 23 bits, unsigned
- Easiest to think in terms of scientific notation
- Beware of rounding when using floats
- Floating point operations are very resource-intensive
 - Use integers if at all possible
- Double precision
 - Uses 64 bits – more for exponent and fraction
 - Even more resource-intensive operations

Programming Basics - Constants

- Constants are very useful for recording the meaning of values that don't change while the program is running, like how your hardware is set up
- Constants are a convenient way to use the same value over and over, but if you need to change it, you only have to change it in one place
- Constants are declared using the syntax `#define CONSTANT_NAME value`
 - `#define NUM_INPUTS (2)`
- Literal constants evaluate to their own value. Symbolic constants evaluate to the value given in the `#define` statement.
- Example
 - If you see `a = 2;` in a program, you're going to wonder, "Two? What does two mean here?"
 - If you see `a = NUM_INPUTS;` you know what the value means

Programming Basics – Statements and Expressions

- Statements are a complete set of instructions to tell the computer to do something
 - Add 2 and 3 and put the result in the variable x
 - `x = 2 + 3;`
 - Call a function and put the result in the variable x
 - `x = product(2, 3);`
- An expression is anything that evaluates to a numeric value
 - Simple expression: one number or a variable
 - Complex expression: simple expressions joined by operators

Programming Basics - Operators

- Operators instruct C to perform some operation on one or more operands
 - Assignment operator
 - Mathematical operators
 - Relational operators
 - Logical operators

Programming Basics - Operators

- Assignment operator: =
 - Assigns the value of one operand to the other
 - $x = y;$ is not “x is equal to y”, but “assign the value of y to x”
 - General form: *variable = expression;*
 - When executed, *expression* is evaluated and its result is assigned to *variable*

Programming Basics - Operators

- Mathematical operators
 - Increment: ++
 - $x++;$ is equivalent to $x = x + 1;$
 - Decrement: --
 - $x--;$ is equivalent to $x = x - 1;$
- Binary operators
 - + is addition: add two operands – $x + y$
 - - is subtraction: subtract two operands – $x - y$
 - * is multiplication: multiply two operands – $x * y$
 - / is division: divides first operand by second operand – x / y
 - % is modulus: gives remainder when the first operand is divided by the second operand – $x \% y$

Programming Basics - Operators

- Operators are evaluated based on precedence
 - ++ and -- are evaluated first
 - *, / and % are evaluated second
 - + and – are evaluated third
- If an expression contains more than one operator of the same precedence, the operators are performed left to right in the order they appear on the line
- Parenthesis can be used to change the order
- Example 1:
 - $x = 4 + 5 * 3, x=19$
 - $x = (4 + 5) * 3, x=27$
- Example 2:
 - $X = 6 / 3 * 2, x = 4$
 - $X = 6 / (3 * 2), x = 1$

Programming Basics – Relational Operators

- Used to compare expressions
- Evaluate to 1 (true) or 0 (false)
- Equal: ==
 - $x == y$ means “Is x equal to y?”
- Greater than: >
 - $x > y$ means “Is x greater than y?”
- Less than: <
 - $x < y$ means “Is x less than y?”
- Greater than or equal to: >=
 - $x >= y$ means “Is x greater than or equal to y?”
- Less than or equal to: <=
 - $x <= y$ means “Is x less than or equal to y?”
- Not equal: !=
 - $x != y$ means “Is x not equal to y?”

Operations and Types

- Make sure your types are big enough to support the result you want
 - Add two large 16 bit values and assign to a 16 bit variable – what happens?
- Make sure your types reflect the “signed” convention you want
 - An unsigned type cannot be less than zero
 - What happens when you do math with signed operands and assign the result to an unsigned variable?
- Make sure your types reflect the precision you need
 - Think very carefully when mixing integers and floats

Programming Basics – Boolean Logic

- Boolean logic allows programmers to compare the output of an expression
- Boolean logic is used to examine the work a program is doing and control the program's flow
- There are several basic comparisons
 - AND – output is true only if all inputs are true
 - OR – output is true if at least one input is true
 - NOT – output is true if input is false

Programming Basics – Boolean Logic

- Truth tables demonstrate these basic functions

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

Output is true only when all inputs are true

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

Output is true when any input is true

x	NOT x
0	1
1	0

Output is the opposite of input

Programming Basics – Logical Operators

- Used to combine relational operators or expressions
- Any number of relational operators and expressions can be combined to produce a result
- AND: `&& x && y` is true if both x and y are true
- OR: `|| x || y` is true if either x or y is true
- NOT: `! x` is true if x is false

Exercises - Operators

- $x = (4 + 5) > (4 * 5)$
 - $x = (9) > (20)$
 - Is 9 greater than 20? No, so $x = 0$
- $y = (8 >= 7) \&\& (25 == 2)$
 - 8 is greater than 7, so that's true
 - 25 and 2 are not equal, so that's false
 - If we AND those expressions, we get false, so $y = 0$
- $z = (20 \% 7) > 4$
 - $z = (6) > 4$
 - Is 6 greater than 4? Yes, so $z = 1$

Functions

- What are they?
 - Functions are a set of operations grouped so you can ask the computer to perform them over and over
- Why are they useful?
 - Functions let you organize and re-use code. If you find yourself writing code for the same sequence of operations more than once, you should put those operations in a function and call it from everywhere you need those steps

Functions

```
/* Program to calculate the product of two numbers */
#include <stdio.h>

int main()
{
    int a, b, c;

    /* Input the first number */
    printf( "Enter a number between 1 and 100: " );
    scanf( "%d", &a );


    /* Input the second number */
    printf( "Enter another number between 1 and 100: " );
    scanf( "%d", &b );

    /* Calculate and display product */
    c = product( a, b );
    printf( "%d times %d = %d\n", a, b, c );

    return 0;
}

/* Function returns the product of its two arguments */
int product( int x, int y )
{
    return( x * y );
}
```

It's called here.



Remember this function?



Functions – The Prototype

Return type –
what type of
data does this
function
return?

Function name

Arguments –
what kind of data
does this function
accept, and how
do we use that
data inside the
function?

```
int product( int x, int y )  
{  
    return( x * y );  
}
```

Functions

```
int product( int x, int y )  
{  
    return( x * y );  
}
```

Function
scope – the
statements
under the
prototype
encased in
curly braces

So what does it do?

- The *arguments* x and y are multiplied
- The result is *returned* to the calling statement

Functions

```
/* Program to calculate the product of two numbers */
#include <stdio.h>

int main()
{
    int a, b, c;

    /* Input the first number */
    printf( "Enter a number between 1 and 100: " );
    scanf( "%d", &a );

    /* Input the second number */
    printf( "Enter another number between 1 and 100: " );
    scanf( "%d", &b );

    /* Calculate and display product */
    c = product( a, b );
    printf( "%d times %d = %d\n", a, b, c );

    return 0;
}

/* Function returns the product of its two arguments */
int product( int x, int y )
{
    return( x * y );
}
```

The *variable* "c" is declared as *type* "int" here...

...so we can *assign* the result of the *function* "product" to the *variable* "c"

...which is the same *return type* as our function...

Why Use Functions?

- Functions allow you to create one block of code that performs similar work over and over again
 - The arguments let you operate on different input
- Functions allow you to break up your code so it is easier to read and maintain
 - You can “hide” the details of a particular operation

Functions - Usage

- Functions can only *return* one piece of information
 - The *prototype* declared what that one thing is
 - There are techniques for getting more than one piece of information from a function
- Functions should only return primitive types
 - Returning larger amounts of data is handled through other means

Variable Scope

- *Scope* defines which program elements are visible from which functions
- *Global variables* are visible by an entire program
 - Generally considered bad practice to use global variables
 - They are sometimes necessary
- *Static variables* are generally visible by an entire file, but only that one file
 - Use is accepted, but they should be used with care
- *Local variables* are visible only to the current function
 - Not visible to functions called from the current function
 - Use *function arguments* to pass data to function calls

Variable Scope

```
/* Program to calculate the product of two numbers */
#include <stdio.h>

int main()
{
    int a, b, c;

    /* Input the first number */
    printf( "Enter a number between 1 and 100: " );
    scanf( "%d", &a );

    /* Input the second number */
    printf( "Enter another number between 1 and 100: " );
    scanf( "%d", &b );

    /* Calculate and display product */
    c = product( a, b );
    printf( "%d times %d = %d\n", a, b, c );

    return 0;
}

/* Function returns the product of its two arguments */
int product( int x, int y )
{
    return( x * y );
}
```

These
variables are
not visible...

...to this
function
because of
scope

Variable Scope – Static and Global Variables

```
#include <stdio.h>
```

```
/* Global variable */  
int a;
```

This *global variable* is visible to the entire program

Both of these are *declared* outside the scope of any function

```
/* Static variable */  
static int b;
```

This *static variable* is visible only to this file

```
/* Main entry point */  
int main()  
{  
...  
}
```

Program Flow Control

- The true power of a programming language is gathering *input* and making decisions based on that data
- You can use what you just learned about operators to have the computer do what you want it to do under given conditions
- This is usually called “program flow control” or “execution flow control”

Let's Make Breakfast!

- You're in bed, and you just woke up. You're feeling pretty hungry.
 - What steps do you take to get to the kitchen and make breakfast?
 - What decisions do you make along the way?
 - How would you program a computer to go through the same thought process?
 - Which parts of the process are time-based?

Let's Make Breakfast!

- You would:
 - Get out of bed
 - Walk to the kitchen
 - Check the fridge for eggs
 - What if there are no eggs? Do you want something else?
 - Check the pantry for a frying pan and spatula
 - What if there are no clean dishes?
 - Prepare the eggs
 - How do you want the eggs cooked?
 - Eat the eggs
 - Do you want more eggs?
 - Clean up the dishes and put away the eggs

Pseudo Code

- Pseudo code is a common programming tool to work out the logical flow of a program before spending the effort to write real code.
- Writing real code is much easier if you already have a solid idea of how your program flow should work.
- It's a programmer's method to "measure twice, cut once"

Let's Make Breakfast – Pseudo Code

```
void start_day()
{
    hungry = TRUE;
    full = FALSE;
    get_out_of_bed();
    walk_to( kitchen );
    open( fridge );
    open( pantry );
    if( ( contains( fridge, EGGS ) == TRUE )
    && ( contains( pantry, PLATE ) == TRUE )
    && ( contains( pantry, FRY_PAN ) == TRUE )
    && ( contains( pantry, SPATULA ) == TRUE ))
    {
        plate = get( PLATE );
        pan = get( FRY_PAN );
        utensil = get( SPATULA );
        while( hungry )
        {
            cook( EGGS );
            place( EGGS, plate );
            full = eatFrom( plate );
            hungry = !full;
        }
        clean_kitchen( pantry, cupboard, plate, pan, utensil );
    }
    else
    {
        clean_kitchen( pantry, cupboard, NULL, NULL, NULL );
        walk_to( bedroom );
        get_in_bed();
        sleep( ONE_HOUR );
    }
}
```

Open up everywhere
we need to look

Check for everything we need – don't make
eggs if we're missing something

Eat until we're full

Clean up so Mom
doesn't find a mess!

If we didn't find everything we
need, clean up anything we
took out and go back to bed.
Maybe it'll be there later.

Program Flow Control – Conditional Statements

- The “choices” your program makes are called “conditionals”
- The most common type of conditional is the if-else statement
- *if* some condition is true, execute one set of statements
- *else*, execute another set of statements
- An *if* clause can be present without an *else* clause
- Example – if x and y are equal, do_stuff() will be run. However, if x and y are not equal, other_stuff() will be run.

```
if( x == y )
{
    do_stuff();
}
else
{
    other_stuff();
}
```

Let's Make Breakfast – Pseudo Code

```
void start_day()
{
    hungry = TRUE;
    full = FALSE;
    get_out_of_bed();
    walk_to( kitchen );
    open( fridge );
    open( pantry );
    if( ( contains( fridge, EGGS ) == TRUE )
    && ( contains( pantry, PLATE ) == TRUE )
    && ( contains( pantry, FRY_PAN ) == TRUE )
    && ( contains( pantry, SPATULA ) == TRUE ) )
    {
        plate = get( PLATE );
        pan = get( FRY_PAN );
        utensil = get( SPATULA );
        while( hungry )
        {
            cook( EGGS );
            place( EGGS, plate );
            full = eatFrom( plate );
            hungry = !full;
        }
        clean_kitchen( pantry, cupboard, plate, pan, utensil );
    }
    else
    {
        clean_kitchen( pantry, cupboard, NULL, NULL, NULL );
        walk_to( bedroom );
        get_in_bed();
        sleep( ONE_HOUR );
    }
}
```

This if statement checks the result of everything between the first and last parenthesis. In this case, we want the function “contains” to be called 4 times, and we’ll check to see that all 4 results are TRUE

This group of code will execute if all four needed items are available. This is called an “if” case.

This group of code will execute if at least one of the needed items is missing. This is also called an “else” case. The ‘else’ case executes when the expression in the “if” case evaluates to FALSE.

Program Flow Control - Loops

- Loops are a method to execute a group of statements over and over until some condition is met
- There are three main types of loops in C
 - While loop
 - Do-While loop
 - For loop

Program Flow Control – While Loops

- While loops execute the code in their scope *while* the condition they're checking is true
- The condition is checked before every iteration of the loop
- If the condition the loop is checking is false when the program gets to the start of the loop for the first time, the code inside the loop will not run.
- The code in the loop will stop executing when the condition the loop checks becomes false

```
while( x )  
    {  
    do_stuff( ) ;  
    }
```

Let's Make Breakfast – while loops

```
void start_day()
{
    hungry = TRUE;
    full = FALSE;
    get_out_of_bed();
    walk_to( kitchen );
    open( fridge );
    open( pantry );
    if( ( contains( fridge, EGGS ) == TRUE )
    && ( contains( pantry, PLATE ) == TRUE )
    && ( contains( pantry, FRY_PAN ) == TRUE )
    && ( contains( pantry, SPATULA ) == TRUE ) )
    {
        plate = get( PLATE );
        pan = get( FRY_PAN );
        utensil = get( SPATULA );
        while( hungry )
        {
            cook( EGGS );
            place( EGGS, plate );
            full = eatFrom( plate );
            hungry = !full;
        }
        clean_kitchen( pantry, cupboard, plate, pan, utensil );
    }
    else
    {
        clean_kitchen( pantry, cupboard, NULL, NULL, NULL );
        walk_to( bedroom );
        get_in_bed();
        sleep( ONE_HOUR );
    }
}
```

How do we know the loop will run? Hint: look at what “hungry” is set to at the top of the function.

Here's our while loop for eating breakfast. What condition are we checking? How does that condition change?

Program Flow Control – do-while loops

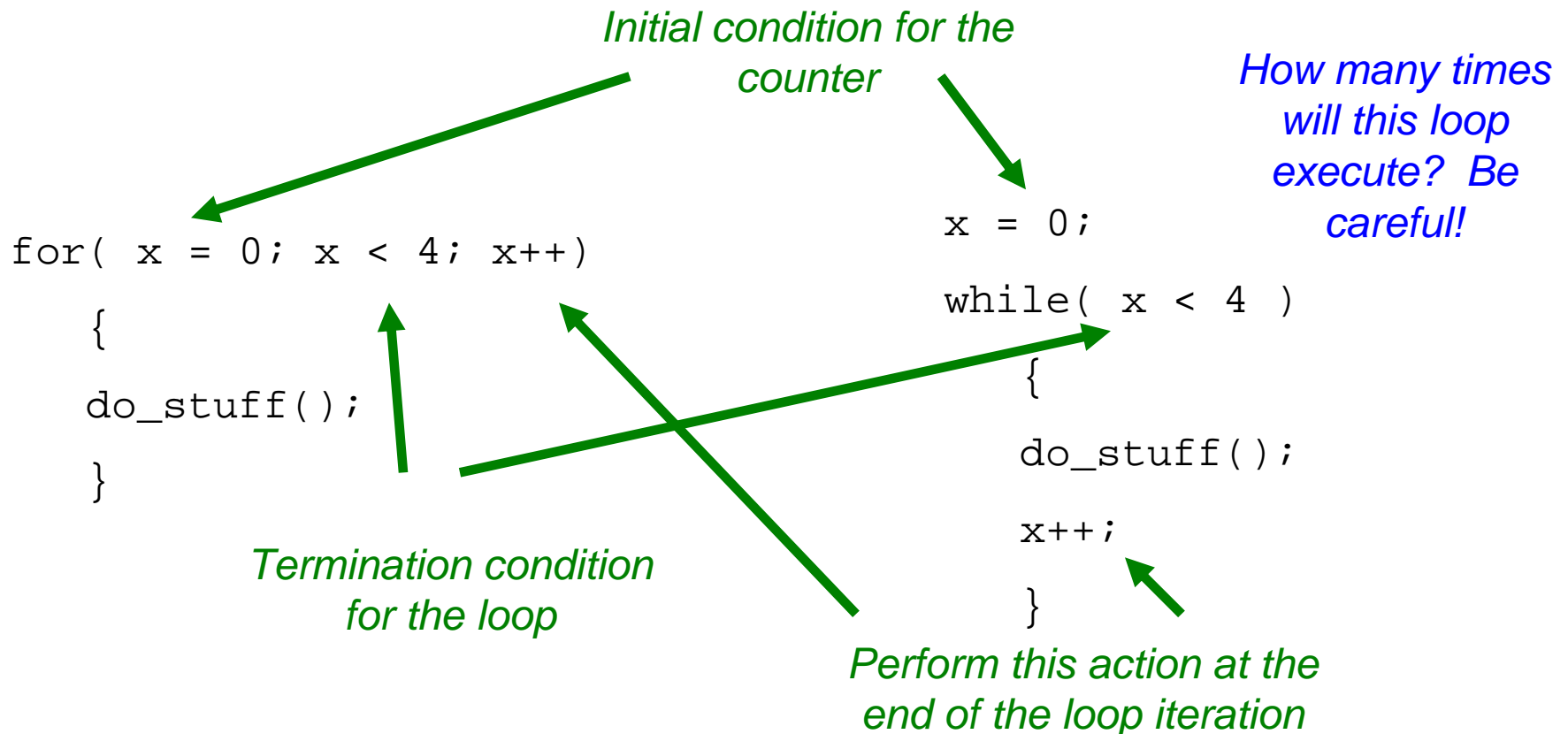
- Do-while loops are like while loops
 - The code in a do-while loop is always executed at least once
 - After that, they work just like a while loop

do

```
{  
do_stuff();  
} while( x );
```

Program Flow Control – for loops

- For loops work on the same basic principle – they execute code until some condition is met
- All for loops can also be written as while loops



Program Flow Control – Variable Scope

- The scope, or visibility, of a variable depends on where it is declared
- A variable declared in a file but outside of any function is *global* – it's visible to any function in your program
- A variable declared inside a function is only visible to that function

Program Flow Control – What Not To Use

- Do not use the goto keyword
 - The goto keyword allows you to jump from one place to another in your code unconditionally
 - Code with goto is extremely hard to read and debug
 - Use if statements instead
- Do not use recursion
 - It is possible to call a function from within itself – this is called recursion
 - This practice can consume a lot of memory – this is bad on embedded systems like robot controllers
 - Almost any recursive function can be handled using loops

Periodic Function Calls

- Most embedded devices (like robots) don't have code that runs once straight through
- Generally, a timer is used to call functions *periodically*, or once per set time period
- The time scales most embedded devices (and most computers) use are usually in the millisecond range ($1/1000^{\text{th}}$ of a second)
 - The “blink of an eye” is usually 300-400 msec
 - Humans generally perceive visual changes around 100 msec and aural changes at 10 msec

Cooking Eggs Periodically

- Let's go back to cooking eggs
 - What do we need to do before we start cooking?
 - What do you need to check while the eggs are cooking?
 - How often do you need to check to make sure the eggs are cooked correctly?
 - Could we miss something if we pick the time period incorrectly?
 - What should we do when we're not checking on the eggs?
 - What do we need to do when we're done cooking?

Cooking Eggs Periodically

You design the function!

More Flow Control

- *If-else* conditions can be used to execute:
 - *One* of a set of options
 - *Each* of several options
- All types of conditionals can be nested
 - Loops can be inside of if-else, which can be inside...

What is the difference?

```
if( a )
{
    do_stuff();
}
else if( b )
{
    other_stuff();
}
else
{
    more_stuff();
}
```

```
if( a )
{
    do_stuff();
}
if( b )
{
    other_stuff();
}
if( c )
{
    more_stuff();
}
```

What is the difference?

```
if( a )
{
    do_stuff();
}
else if( b )
{
    other_stuff();
}
else
{
    more_stuff();
}
```

- The condition “a” is evaluated first
 - If “a” is true, the code block under it will be executed
 - The program will then jump below the “else” block and continue
- If “a” is false, “b” is evaluated
 - If “b” is true, the code block under it will be executed
 - The program will then jump below the “else” block and continue
- If “a” and “b” are both false, the code under the “else” block is executed

What is the difference?

- The condition “a” is evaluated
 - If “a” is true, the code block under it is executed
- The condition “b” is evaluated
 - If “b” is true, the code block under it is executed
- The condition “c” is evaluated
 - If “c” is true, the code block under it is executed
- The evaluation of one condition does not affect the others

```
if( a )
{
    do_stuff();
}
if( b )
{
    other_stuff();
}
if( c )
{
    more_stuff();
}
```

More Flow Control

- An *if* statement can be used without an *else*
- *If* and *else if* can also be used together without an *else*
- An *if* is required to use an *else if* or an *else*

Evaluating Conditions

- Conditions are evaluated in the order in which they appear
- If combining conditions with *and* (&&), the conditions are evaluated until all are found to be TRUE or one is found to be FALSE
- If combining conditions with *or* (||), the conditions are evaluated until one is found to be TRUE or all are found to be FALSE
- These properties can be used to ensure data is valid before it is evaluated

Evaluating Conditionals

```
if( ( contains( fridge, EGGS ) == TRUE )
    && ( contains( pantry, PLATE ) == TRUE )
    && ( contains( pantry, FRY_PAN ) == TRUE )
    && ( contains( pantry, SPATULA ) == TRUE ) )
```

In this case, the function `contains()` will be called to check if there are any eggs in the fridge. The result of that function will be checked for equivalence to `TRUE`. If the fridge contains eggs, the next condition will be checked.

If there are no eggs in the fridge, does it matter what dishes or utensils are in the pantry?

If we don't have eggs, can we still make breakfast?

Evaluating Conditionals

```
if( ( contains( fridge, EGGS ) == TRUE      )
    || ( ( contains( fridge, MILK ) == TRUE    )
         && ( contains( pantry, CEREAL ) == TRUE ) ) )
{
  makeBreakfast();
}
```

How will this be evaluated?

1. First, the *fridge* is checked for *eggs*
 - If there are eggs, we make breakfast
 - If there are no eggs, we keep going.
2. Assuming no eggs, the *fridge* is checked for *milk*
 - If there is no milk, we stop. We don't want cereal without milk.
 - If there is milk, we keep going.
3. Assuming there is milk, the *pantry* is checked for *cereal*
 - If there is cereal, we make breakfast

More on Periodic Processing

- Understand the rate at which your periodic function is called
 - What do you want to do every time?
 - What do you want to do at some other period?
- Be sure the tasks you perform can be accomplished within one time period
 - An alternative is to perform different tasks in different iterations
 - This approach is called a *state machine*

Driving With the Compass – One Solution

```
StudentPeriodic( void )
{
#define FULL_FWD ( 1.0f )           //motor speed for full forward
#define TURN_FWD ( 0.5f )          //motor speed for turning
#define ANGLE_LOW ( 44.0f )         //low angle limit
#define ANGLE_HIGH ( 46.0f )       //high angle limit

float angle;
angle = m_compass->getAngle();
if( angle < ANGLE_LOW )             //if angle is too low, turn right
    {
    SetLeftMotor( FULL_FWD );
    SetRightMotor( TURN_FWD );
    }
else if( angle > ANGLE_HIGH )       //otherwise if angle is too high, turn left
    {
    SetLeftMotor( TURN_FWD );
    SetRightMotor( FULL_FWD );
    }
else                                 //otherwise, just go forward
    {
    SetLeftMotor( FULL_FWD );
    SetRightMotor( FULL_FWD );
    }
}
```

Why Does That Solution Work?

- The StudentPeriodic() function is called periodically
 - No loops are necessary because the compass value is checked and the drive train adjusted every time period
- The angle is checked for *bounds*, not *equivalence*
 - Floating point equivalence (==) is nearly impossible
- The angle selected bounds produce the desired result without constantly changing motor speeds
 - Real motors don't change speed instantly
 - Would looser bounds be OK? What about tighter bounds?

Why Does That Solution Work?

- If the angle is below the lower bound, turn right
 - Set the right wheel to a slower speed
 - Set the left wheel to full speed
- Else if the angle is above the upper bound, turn left
 - Set the right wheel to full speed
 - Set the left wheel to a slower speed
- Else, go straight
 - If the compass angle is not too small or too big, it must be within the selected bounds
 - Set both wheels to full speed

What Not To Do

- Don't loop inside the periodic function and wait for the angle to change
 - The periodic function will try to run the loops each time through
 - Staying in the loop prevents the other simulator code from running, so the simulated robot won't move
- Don't wrap the desired logic in a loop
 - The periodic nature of the function will call the logic over and over again

A Bit About State Machines

- A *state machine* means that the program will act based on its current *state*
- This can be used to cause the periodic function to act differently on subsequent calls
 - You can perform different actions in different time slices by maintaining state information
 - This can be useful when you have several tasks to perform that all take your entire time slice but they don't need to be performed at the fastest rate possible

Structures

- Structures are a way to group data
- Declaring a variable using a structure type lets you declare the variable once but get all the data allocated
- Structures are most useful when you need the same set of data in several places

Using Structures

- Structures are defined using the *typedef* keyword
- Structures can contain primitive types or other structures
- Structure members are assigned or used by *dereferencing* the desired member
- Structures cannot be assigned to one another like primitives
 - A memory copy must be used to copy an entire structure

Using Structures

```
//declaring the type
typedef struct
{
    char x;
    char y;
    int a;
    int b;
    float c;
} demo_type;

//declare the variable - same syntax as primitives
demo_type demo;

//initialize the variable - can't use an assignment
memset( &demo, 0, sizeof( demo ) );

//set values with dereferencing
demo.x = 'j';
demo.a = 125;
demo.c = 55.2f;
```

Where Is My Data?

- Every declared piece of data is stored at an address
- You can access that address and use it in your program
- You can use the address for access to large structures in functions
- This functionality is made possible with the use of *pointers*

Pointers

- *Pointers* give you the address of a variable
- Pointers can be used to access the data directly through *dereferencing*
- Pointers allow you to pass the location of data around your program without copying it to a new location every time you want to use it
- Pointers have types just like other variables
 - A pointer of type `int` means the data at that address will be interpreted as an `int`

Operations Using Pointers

- The * operator *dereferences* a pointer which gives you access to the *value* at that memory location
 - This operator is generally used in front of a pointer variable
 - *p_a dereferences the pointer p_a to obtain the value at that address
- The & operator gives you the *address* of a variable which can be used with pointers
 - This operator is generally used in front of a variable
 - &a gives the address of the variable a and can be used to set the value of a pointer

Pointers

```
int a, b;           //declare variables
int * p_a;         //declare a pointer

//initialize variables
a = 6;
b = 7;

//set the value of p_a to the address of a
p_a = &a;

//output the value of p_a and the value at p_a
printf( "%d is at address %08x\n", *p_a, p_a );

//change the value at p_a
*p_a = b;

//output the value of p_a and the value at p_a
printf( "%d is at address %08x\n", *p_a, p_a );
```

Classes

- Classes are an *object-oriented* concept
- Classes allow you to group data and functionality in to a package that can easily be reused throughout your program
- Those packages are typically called *objects*

Classes

- Classes can contain variables and functions
- Classes are built with a *constructor*
 - Constructors are called when you declare a variable with an object type
 - Constructors typically initialize an object's variables and perform any other necessary setup
- Classes are demolished with a *destructor*
 - Destructors typically free any memory or other resources that the object claimed while it was in use

Classes

- Class items can be *public* or *private*
 - Public items can be accessed directly, similar to global variables
 - Private items cannot be accessed directly
 - Private variables are often accessed using functions called *accessor methods*

Using Classes

- Creating a new class

```
robot * bender = new robot;
```

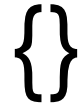
- Accessing class variables

```
bender->best_friend = fry;
```

- Accessing class functions

```
bender->bend( girder );
```

Common Mistakes – Curly Braces



- Curly braces out of place
 - Curly braces control the scope of functions, loops and if statements
 - Almost all development environments have a feature where you can find matching curly braces
 - If you put your cursor on an open curly brace, it can find the corresponding close curly brace, or vice versa
 - Use this tool to make sure the code you've written is scoped correctly
 - In or out of loops
 - In or out of if statements
 - Having statements outside of a function's curly braces can also cause compiler errors

Common Mistakes - Syntax

- C syntax is very complex – it's easy to make a mistake (even for professionals!)
- The compiler will catch syntax errors – it won't know what to do
- Use the line number given by the compiler to track down your mistake
 - Check for a semicolon at the end of the line – should one be there?
 - Do you have too many open or close parenthesis?
 - Are all your operators valid?
 - Are all your variables declared in the right scope?
 - Check other lines above the one indicated – some syntax errors will show up on one line, but the error is really a line or two up

Asking For Help

- If you really get stuck, ask for help!
- When you ask for help, make sure you ask a good question
 - Good questions contain enough information for the person you're asking to help you
 - Good questions show you've made an honest effort to solve the problem – do this by including the steps you've already taken in your question
 - Good questions make the best use of your time and your mentor's time, and will often get better, faster responses
- Bad questions:
 - “I don't get it.”
 - “I'm getting a syntax error.”
- Good question:
 - “In the file foo.c, I'm trying to call the function bar(), but the compiler says bar() is undefined. I checked the scope, and it looks OK. What am I missing?”