



Introduction to C/C++ Programming

December 17, 2011

Presented by Garmin International

Agenda

Getting started with C/C++

- What is it and how do I use it?

Example program

Basics of programming

- Boolean logic
- Data types
- Statements, expressions, operations
- Variables and constants
- Basic arithmetic

Flow control

- If/else, while loops, for loops, variable scope
- Functions
- Bad practices: goto, recursion

Common mistakes

- Syntax, scope

C or C++ What is the Difference?

- **Bell Laboratories**
- **C came first**
- **Use ++ to increment a value in C**
 - ...so C++ is like C + 1
- **C++ added classes and other “object orientation” constructs**
- **C tends to be more deterministic**
- **With care C++ can be deterministic**

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 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 – convert *C code* in to *machine code*
- Assembler – convert *assembly code* into *machine code*
- Linker – take all the machine code (*object code*) and put it together with *libraries* to form your program

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**

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
- *double* – high precision floating point number, used to represent values like 32.05

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?

1,764,018,660 in decimal

0110 1001 0010 0100 1100 1001 1110 0100

Sign Bit

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!

- Avoid `==` with floats.

Floating point operations are 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)`

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:

- $X = 6 / 3 * 2, x = 4$
- $X = 6 / (3 * 2), x = 1$

Use Parenthesis to avoid needing to remember the precedence!

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 each of 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

Output is true when any input is true

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

Output is the opposite of input

x	NOT x
0	1
1	0

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

Example - 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;
}
```

It's called here.



Remember this function?



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

What are the other functions in this code snippet?

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;
}
```

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*...

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

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
- “Hide”? What is the secret?

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

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

This *static variable* is visible only to this file

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

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

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”

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();
}
```

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

```
x = 1;
```

```
while( x )
```

```
{
```

```
do_stuff();
```

```
}
```

How would you stop this loop from executing?

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

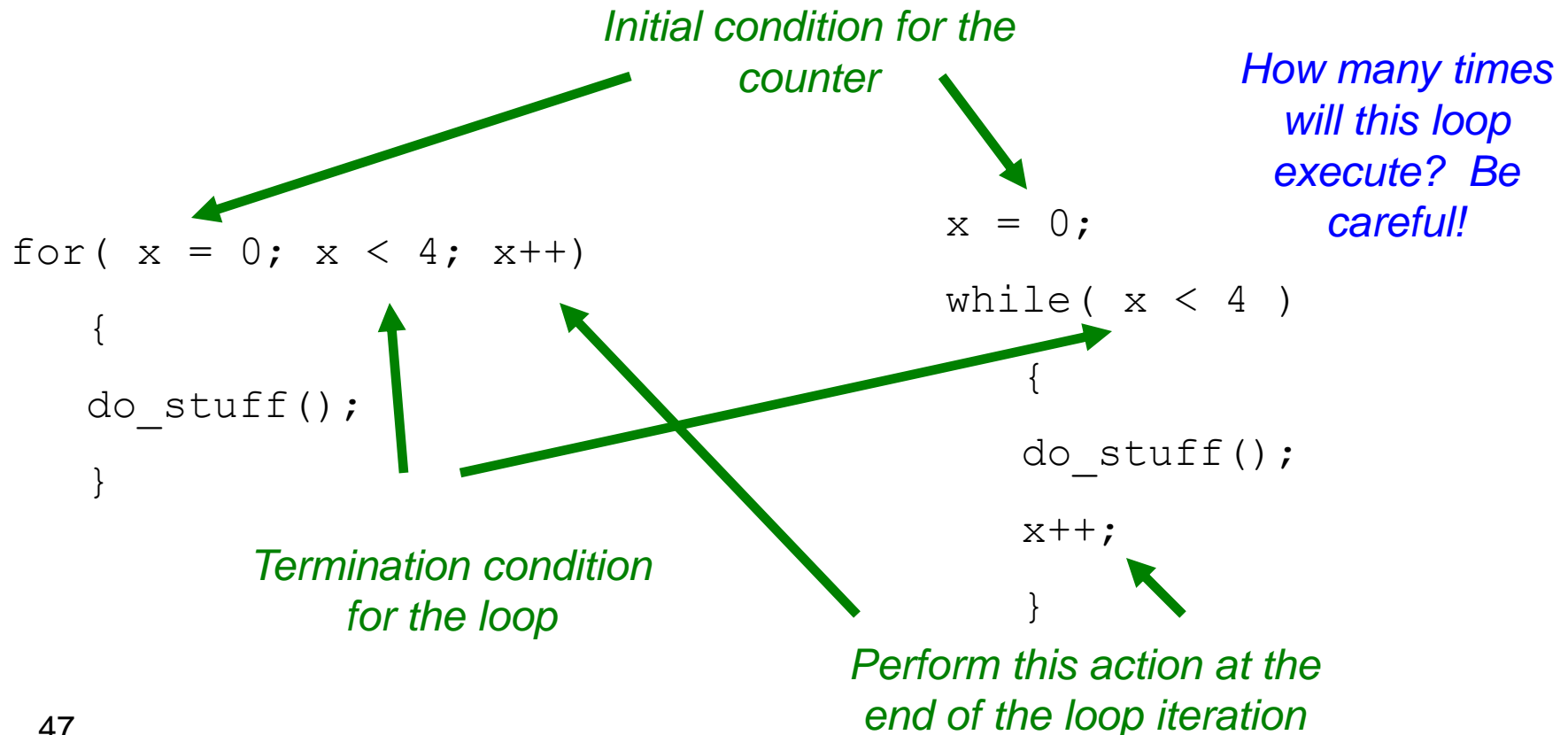
do

```
{  
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

For loops can also be written as while loops



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 (1/1000th 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

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();
}
```

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*

What Not To Do

Don't loop inside the periodic function and wait for something to change

- The periodic function will try to run the loops each time through
 - Staying in the loop prevents the other code from running, so the robot will NOT respond well

Don't wrap the desired logic in a loop

- The periodic nature of the function will call the logic over and over again

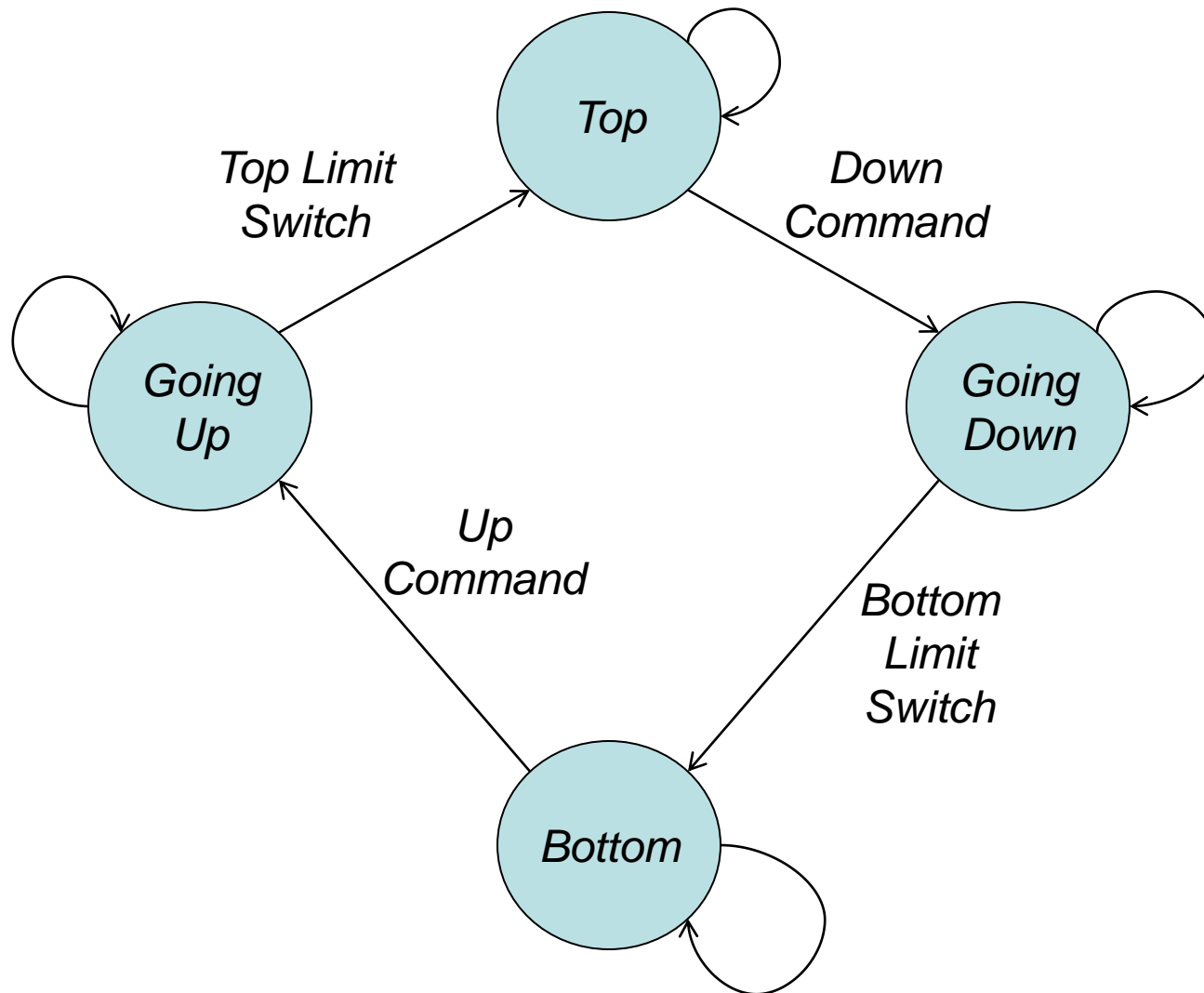
A Bit About State Machines

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

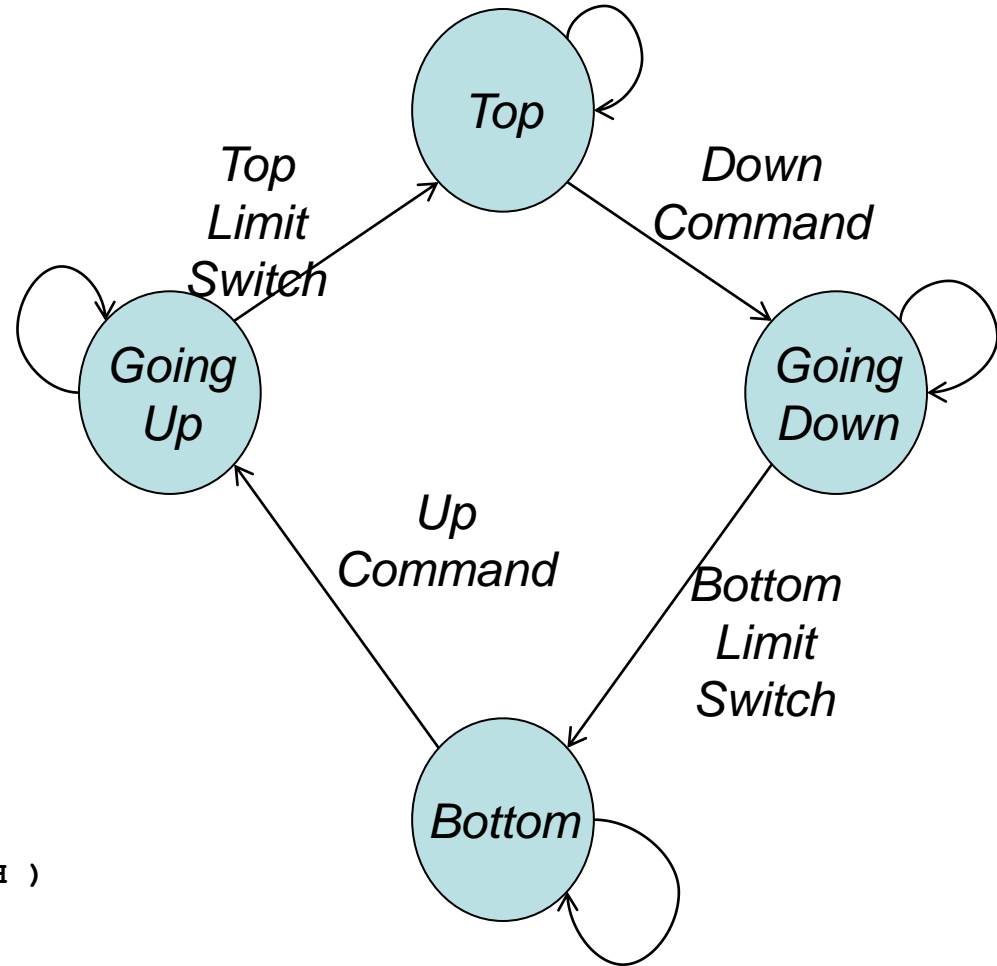
Robot Lift State Machine



Robot Lift State Machine Code

```
StudentPeriodic( void )
{

int command = GetCommand( );
int switches = GetSwitches( );
switch( state )
{
case BOTTOM:
    motor( OFF );
    if( command == GO_UP )
        state = GOING_UP;
    break;
case GOING_UP:
    motor( UP );
    if( switches == TOP_SWITCH )
        state = TOP;
    break;
case TOP:
    motor( OFF );
    if( command == GO_DOWN )
        state = GOING_DOWN;
    break;
case GOING_DOWN:
    motor( DOWN );
    if( switches == BOTTOM_SWITCH )
        state = BOTTOM;
    break;
}
}
```



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 can contain variables and functions

Classes are built with a *constructor*

- Constructors are class functions that are automatically called when you declare a variable with an object type or create one with `new`
- Constructors typically initialize an object's variables and perform any other necessary setup

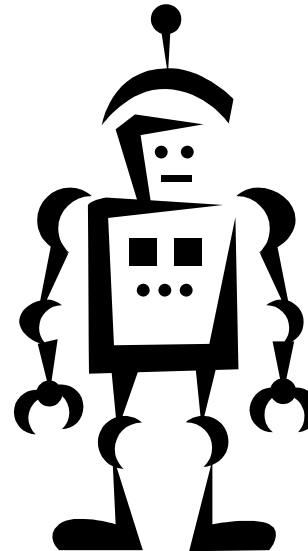
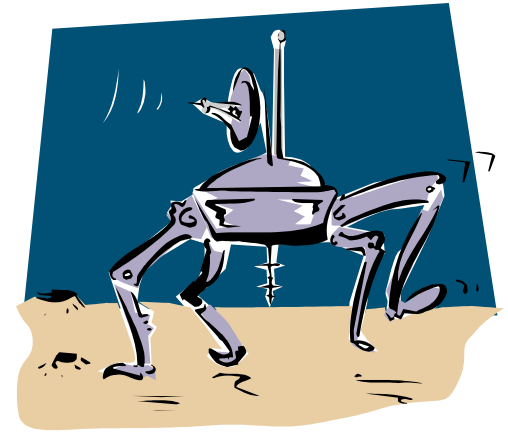
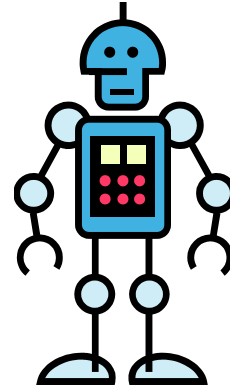
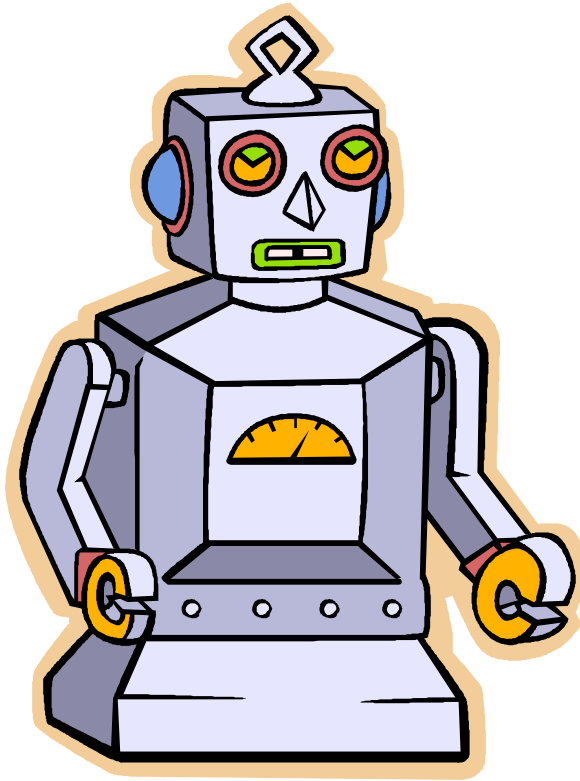
Classes are demolished with a *destructor*

- Destructors are class functions that are automatically called when class object goes out of scope or is removed with `delete`
- Destructors typically free any memory or other resources that the object claimed while it was in use

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*

Let's try some code...



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