ECE 495/595 Lecture Slides

Winter 2017

Instructor: Micho Radovnikovich

Summary and Quick Links

These slides quickly introduce the following C++ concepts:

- Comments (Slide 3)
- Variables (Slide 4)
- Data types (Slide 5)
- Pointers (Slide 6)
- Structures (Slide 8)
- Functions (Slide 10)
- Vectors (Slide 15)
- Program structure (Slide 18)
- Preprocessor directives (Slide 21)
- Header files (Slide 25)
- Namespaces (Slide 27)
- Classes (Slide 32)

It is encouraged to go through a good C++ tutorial at http://www.cplusplus.com/doc/tutorial/

Comments

- There are two ways to make comments in C++.
- Double slash <u>//</u> marks the rest of the line as a comment.
- Slash-asterisk <u>/*</u> starts a multi-line comment and is ended by asterisk-slash <u>*/</u>

```
// This is a comment

/* This is a
   multi-line comment */
```

Variables

- A C++ variable is declared by indicating a type, followed by the name of the variable, and then an optional initialization value:

```
// variable_type variable_name = initial_value;
int my_variable = 4;
```

- Simple variables can be manipulated like in many other programming languages:

```
int a;
int b;

a = 5;
b = a + 10;
```

Data Types

- Integer data types (each have signed and unsigned versions):
  - *char* – 8 bits
  - *short* – 16 bits
  - *int* – 32 bits

- Floating point data types:
  - *single* – 32-bit floating point number encoded in the IEEE 754 standard.
  - *double* – 64-bit floating point number encoded in the IEEE 754 standard.

- C++ has a special *std::string* class type to save the user from awkwardly having to use pointers to *char* data to manage strings.

Pointers

- Pointers don't actually contain data; rather they contain the *address* of where the actual data can be found.
- Pointer types match the type they are pointing to, with the <u>*</u> symbol added on.
- The address of a variable is accessed with the <u>*&*</u> operator:

```
double var; // Actual variable
double* var_pointer; // Pointer to a double value

var_pointer = &var; // Assign address to pointer
```

- To access the data that is being pointed to, a <u>*</u> is placed before the pointer variable:

```
double x = *var_pointer; // Dereference pointer
```

Pointers

http://xkcd.com/138/

Structures

- Data types can be defined in structures:

```
typedef struct{
  double val1;
  int val2;
  std::string str;
} StructTypeName;
```

- Individual structure member data are accessed using the . operator:

```
StructTypeName struct_variable;

struct_variable.val1 = 5.0;
struct_variable.val2 = 1;
struct_variable.str = "hello_world";
```

Structures

- If using a *pointer* to a structure variable, the data are accessed with the <u>-></u> operator:

```
StructTypeName struct_variable; // Actual structure
StructTypeName* struct_pointer; // Pointer to structure

struct_pointer = &struct_variable; // Assign pointer

// Access values with '->' operator
struct_pointer->val1 = 5.0;
struct_pointer->val2 = 1;
struct_pointer->str = "hello_world";
```

Functions

- Functions are defined using the following generic syntax:

```
return_type function_name(<list of arguments>)
{
}
```

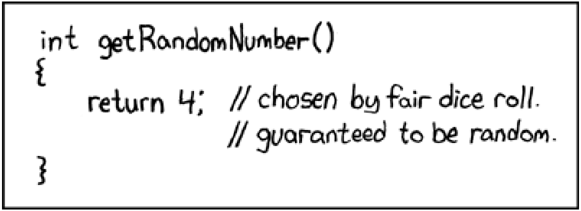
- *return_type* is the data type that is returned from the function.

- Any number of arguments can be passed to a function, where each is separated by a comma. For example:

```
double add(double a, double b)
{
  return (a + b);
}
```

Functions

- Functions without arguments just have empty parentheses:



http://xkcd.com/221/

Functions

- Functions can also be defined with a *void* return data type:

```
void func()
{
  /* This function does nothing
     and returns nothing. */
}
```

- Void functions have no return value at all.

Functions

- Variables declared within a function are *local* variables and cannot be accessed from any other function.
- Any variables declared within an if block or loop can only be accessed within that block or loop.
- Where a variable is able to be accessed is called its *scope*.

Functions

▷ An example illustrating the scope of local variables

```
void example_function()
{
  int val1 = 10; /* This variable is accessible anywhere in this
                    function, but not from any other function */

  if (val1 > 3){
    int val2; /* This variable is only accessible
                 in this 'if' block */
  }else{
    int val3; // Same with this variable
  }

  for (int i = 0; i < 10; i++){
    int val3 = 0; /* This variable can only be used
                     in this for loop */
  }
}
```

Vectors

- Vectors are dynamically sized arrays, declared as a *std::vector*.

- The type of the individual elements must be specified within <> when the vector is declared:

```
std::vector<int> int_vect;
std::vector<double> float_vect;
std::vector<SomeStructure> vector_of_structs;
```

- Individual data elements are accessed like normal arrays with the *[]* operator.

```
int one_element = int_vect[3];
```

Vectors

- Vectors are initially empty by default. Their size can be set when it is declared though:

```
std::vector<int> vect(5); // Initialize vector with 5 elements
std::vector<int> vect(5, 0); // Initialize vector with zeros
```

- A vector's size can also be set directly with the *resize()* method:

```
vect.resize(6); // Set number of elements to 6
vect.resize(6, 0); // ... and also set their values to zero
```

- Single values can be added on using the *push_back()* method:

```
vect.push_back(2); // Adds a new value of 2 to the end
```

Vectors

- The *size()* method returns the current number of elements in the array.

- There are many more ways of manipulating the data within a vector. For full details, see the documentation.

- Be careful! If you try to access an element of a vector that doesn't exist, your program will crash with a *segmentation fault*.

```
std::vector<int> vect; // Empty vector
int one_element = vect[1]; // This will crash!
```

Program Structure

- All C++ programs have a function called *main* that is run when the program starts.

```
int main(int argc, char** argv)
{
  /*
  'argc' is the number of arguments passed to the function
  'argv' is an array of strings; one for each argument
  */

  // <Program code goes here>

  return 0; // The program ends here
}
```

Program Structure

- In addition to the main function, other functions can be defined along with it in the same file:

```
int getRandomNumber()
{
  return 4;
}

int main(int argc, char** argv)
{
  int x = getRandomNumber();

  return 0;
}
```

- Functions must be defined *before* the functions they are called from.

Program Structure

- Global variables are declared outside any functions, typically at the top of the file:

```
double global_var;

void setTo42()
{
  global_var = 42.0;
}

int main(int argc, char** argv)
{
  global_var = 9001.0; // This sets the global variable
  setTo42(); // This function will also set the global variable

  return 0;
}
```

Pre-processor Directives

- Pre-processor directives are commands that affect how the program is compiled, as opposed to being actual program code.
- All preprocessor directive commands begin with a # symbol.

Pre-processor Directives

- *#define* – Defines a macro. Replaces all instances of the macro with its substitution text. This can be used to make code more readable:

```
#define X a_long_variable_name.signals[0].val[0]

// This macro is useful in this case:
int y = X * (X + 5) - 2 * X;
```

- or to perform a simple function:

```
#define ADD(a,b) (a+b)

double x = 5;
double y = 9;
double sum = ADD(x,y);
```

Pre-processor Directives

- *#if*, *#else* and *#endif* – Used to compile pieces of code if certain conditions are met:

```
#define CONDITION 1

#if CONDITION
  // <Code here will be compiled>
#else
  // <Code here will NOT be compiled>
#endif
```

- Any non-zero value will satisfy the *#if* condition.

Preprocessor Directives

- *#ifdef* – Instead of evaluating an expression, this will only compile code if a specific macro has been defined with *#define*.

- *#ifndef* – The opposite... Code is compiled if the macro has *not* already been defined.

Header Files

- Header files are used to interface C++ functions from different source files.
- Header files typically don't contain any code; instead they contain the definitions of functions, types, etc. that are actually implemented in a source file.
- Header files are *included* into a source file to provide these definitions to the functions implemented there.

Including Header Files

- Header files are included at the top of C++ source files.

```
#include "some_header_file.h"

int main(int argc, char** argv)
{
  // <Program code goes here>
}
```

- Including a header file is exactly like copying the text of the header file and pasting it where you type *#include*.

Namespaces

- Variable types, functions, classes and other things can be defined within *namespaces*.
- A benefit of namespaces is to modularize code, and to avoid potential name ambiguity if a function or class of the same name is defined.
- Functions and classes written for use in ROS are typically defined within a namespace with the same name as the ROS package they are a part of.

Namespaces

- Code and header file content are defined in a namespace by placing it within a namespace block:

```
namespace my_namespace{

  /* Anything declared here is in the namespace 'my_namespace' */

}
```

Namespaces

- For example, we can define a structure type and a function within this namespace:

```
namespace my_namespace{

  typedef struct{
    double a;
    int b;
  } StructureType;

  void function_name(double var);

}
```

Namespaces

- Elsewhere in code, we can then refer to these using their names, preceded by the namespace name and the :: operator:

```
// Declare a structure variable
my_namespace::StructureType structure_var;
structure_var.a = 4.5;
structuer_var.b = 1;

// Call a function within the namespace
my_namespace::function_name(4);
```

Namespaces

- The *using* keyword helps avoid having to include the namespace name and *::* operator whenever something from that namespace is used.

```
using namespace my_namespace;

my_namespace::StructureType structure_var; // This still works
StructureType structure_var; // This is equivalent
```

Classes

- In ROS and C++ in general, classes are used extensively to create modular code that can be easily used in many different programs.
- In ROS, classes are conventionally defined in a namespace matching the name of the package they are a part of.
- Classes are usually defined using both a header file (*.h*) and a source file (*.cpp*).
- The header file provides the class property and method definitions, and the source file actually implements the methods.

Classes

- In general, every class must have a constructor method and a destructor method.
- These functions carry the same name as the class itself, with the destructor having a tilde ($\underline{\sim}$) in front of its name.
- Variables defined within the class are called *properties*. Functions defined within the class are called *methods*.
- Properties and methods that can be called from the code that instantiated the class are declared as *public*.
- Properties and methods that can only be called from the class methods are declared as *private*.

Classes (Example Header File)

```
namespace ros_package_name{

class ClassName{
  public:
    ClassName(); // This is the constructor
    ~ClassName(); // This is the destructor.

    void method1(int arg); /* This function can be called
                               by the instantiating code */

    double property1; /* This property can be accessed
                          by the instantiating code */
  private:
    double method2(); /* This function can only be called by
                          other class methods */
    int property2; /* This property can only be accessed by
                       class methods */
};
}
```

Classes

- Classes can be instantiated within C++ programs:

```
#include "ClassName.h"

int main(int argc, char** argv)
{
  /* Instantiate the class 'ClassName' from the
     namespace 'ros_package_name' */
  ros_package_name::ClassName instance;

  instance.method1(3); // Call the public method 'method1'
  double var1 = instance.property1; /* Read the public property
                                       'property1' */

  double var2 = instance.method2(); /* This will not compile,
                                       since we are trying to call
                                       a private method */
}
```

Classes

- Class pointers can be declared globally, then initialized using the *new* operator:

```
#include "ClassName.h"

ros_package_name::ClassName* class_pointer;

int main(int argc, char** argv){
  // Instantiate the class using the 'new' operator
  class_pointer = new ros_package_name::ClassName();

  class_pointer->method1(3); // Call the public method 'method1'
  double val1 = class_pointer->property1; /* Read the public
                                          property 'property1' */
}
```

- Keep in mind that like structures, when dealing with class pointers, the methods and properties are accessed with the -> operator instead of the .

Classes

- Here is an example of a class whose constructor has arguments:

```cpp
class ClassName{
  public:
    ClassName(int a, int b){ // Constructor with arguments.
      // Set private class properties with arguments
      a_ = a;
      b_ = b;
    }

    int sum(){ return a_ + b_; }

  private:
    // ROS convention is to put underscores
    // after property names
    int a_;
    int b_;
};
```

Classes

- To instantiate this class with constructor arguments:

```
ClassName class_instance(5, 6); // Instantiate class and
                                // pass arguments to construtor

int sum_val = class_instance.sum(); // Returns the sum of the
                                    // two class properties
```

- The *sum_val* variable would contain a value of 11 in this example.