C++ Programming Guide

Written by Bryce Summers (BryceSummers.com)

Last Updated: August 7, 2015

Contents

1 Overview

This work of nonfiction has been written for those who have some experience with languages with C like syntax and would like a guide to some of the foundational features that a programmer might expect. I have taken the liberty of collecting those bits of information together as a convenience to others.

2 Class Structure

C++ allows a programmer to frolic in the joyful experience that is object oriented programming. I have decided to begin this document by talking about the basic structure of classes and present an example class that will be used in later chapters.

2.1 Example Header File (.h)

```
/* Filename: Example.h */
#pragma once // Only include this header file once!

class Example
{
    public: // Data and methods accessible from anywhere in the program.

        // input is -45 by default if no argument is given.
        Example(int number_in = -45); // Constructor.
        virtual ~Example();            // Destructor.

        int  getNumber(); // Notice the semicolon!
        void setNumber(int n);

    protected: // Same as protected, but also accessible from child (derived) classes.

        virtual int getChildNumber(); // Virtual Method. '=0' for pure virtual.

    private: // DEFAULT, accessible only from a particular instance of this class.

        int number; // Private variable.
};

class subClass : public Example // inheritance.
{
  ... // Constructor and Destructor not shown.
  virtual int getChildNumber();
  private: int childNumber;
}
```

2.2 Example Implementation File (.cpp)

```cpp
/* Filename: Example.cpp */
#include "Example.h"

Example::Example(int n) // Constructor.
{
  number = number_in;
}

Example::~Example() // Destructor.
{
  // Free / Delete any memory here.
}

int Example::getNumber() // Public Method.
{
  return getChildNumber() + number; // number = this -> number.
}

void Example::setNumber(int n)
{
  number = n;
}

int Example::getChildNumber()
{
  return 0;
}

int subClass::getChildNumber()
{
  return childNumber;
}
```

2.3 Constructor

Constructors specify what the initial settings for all of a classes data fields will be. Constructors may take arguments just like functions. Constructors may be implemented in the header file or an associtated implementation file. Constructors are called automatically when an object is first instantiated.

```
Example(int n); // Declaration in Header, implementation in .cpp file.
Example(int n){number = n;}; // Implementation in header.
```

2.4 Destructor

The destructors are called when a stack allocated class goes out of scope or when a delete or free operation is performed.

```
virtual ~Example(); // Destructors are always virtual and have a tilde.
```

For more about the memory management and how it relates to constructors and destructors, please see Section 3.

3 Memory Management

Like C and unlike Java and many newer languages, C++ requires the programmer to partake in manual memory management. This means that the programmer is responsible for freeing memory resources once they are finished. I will discuss 3 different pairs of memory allocation and deallocation procedures in this section. Programmers can also use some features of C++ such as stack based memory allocation, smart pointers, and passing by pointer/reference to manage memory implicitly, therebye handing off the job of freeing themselves of the responsibility of explicitly calling free or delete functions.

3.1 Malloc / Free

Mallocs and frees have been inherited into the C++ language from good old fashion C.

```
int * array = malloc(1000*sizeof(int));
...
free(array);
```

They are used to allocated explicit regions of memory and free them. All memory allocated in this manner will be stored in the heap, so data that has been malloced will be safe as long until free is called. This makes it useful for allocating data that needs to survive outside of the function or class that has allocated it. I would reccomend only using malloc and free for traditional structures such as arrays of data or primitive structures, but to use new and delete when allocating and deallocating classes.

3.2   new / delete

The new and delete operators work like mallocs and frees, but are more user friendly for allocating space for instances of a given class.

```
Example * ex = new Example(5);
...
delete ex;
```

3.3   Stack Based (Static) Allocation

Entire classes can be allocated on the stack, just like primitives such as ints, chars, and floats! As long as you are careful to not pass classes that are gigantic around by value, allocating on the stack will make your life easier. Stack based allocation allows you to not need to worry about explicitly freeing the memory and the allocation itself should be faster than dynamic memory allocation.

```
void foo(){
Example ex; // instantiates the default constructor with the default value (-45 in this
//Example ex = Example(5);
...
// deconstructor is called on 'ex' and 'ex' memory is deallocated.
}
```

Please note that classes themselves can have statically allocated members, provided there is not an infinite descent loop.

```
class smallstruct
{
  int val;
  float val2;
}
class largestruct
{
  smallstruct ss1; // Static allocation.
  int val;
  smallstruct ss2; // Static allocation.
}
int main()
{
  largestruct struc;
  ... // All memory survives until 'struc' is out of scope.
}
```

4 Inheritance

Inheritance can be used to create derived classes that each share some core functionality with the base class, but provide some unique functionality that reflects there own individuality.

```
class subClass : public Example // inheritance.
```

An example of when inheritance is useful is when writing a raytracer for different types of geometry, which each have their onw ray intersection specification.

4.1 Base Classes

Base classes are the general parent class that provide core functionality that may be useful to a number of children with more specific interests.

4.2 Derived / Child Classes

Child classes, also known as derived classes, inherit all of the general functionality from a parent, but then extend the core functionality with custom methods that are more specific to the child.

4.3 Virtual Methods

The C++ `virtual` keyword may be used to label functions as virtual. Virtual methods are those in a parent that provide an implementation, but which may be overridden by a custom implementation in individual child classes.

4.4 Pure Virtual Methods

Pure virtual methods are those that are virtual, but that do not provide an implementation in the base class. These are differentiated from linking errors and missing implementations that are mistakes of the programmer by writing '=0' after the declaration in the header file.

```
virtual int foo() = 0; // '=0' makes it pure, instead of a compile error.
```

Classes with pure virtual functions may not be instantiated, instead the programmer would instantiate child classes that provide implementations for the missing methods. Like normal virtual methods, a base class may make calls to pure virtual methods and the implementation will be filled in by whichever child class a programmer has instantiated. This makes it easy to deal with algorithms that require a large number of particular cases of a general problem, such as the ray geometry intersection test used in Computer Graphics Applications.

4.5   Overriding

A method in a child class is said to *override* a method in a base class if it has the same name as a virtual method in the base class. The child class's implementation of the method will be used in liu of the parent method's implementation.

In the example in Section **??**, the `subClass` class overrides the `getChildNumber` method.

If the `subClass` class wanted to make a call to the parent's `getChildNumber` method, then it would do so as follows:

```
int val = Example::getChildNumber(); // calls base class' function
```

5   Data passing by value, pointer, and reference

Passing by value makes a stack allocated copy. Passing by pointer or reference passes the original memory location.

```
// -- Function Declaration and Implementation.
int add1(int n)     // Value.
{
  n++;
  return n;
}
int add2(int * n); // Pointer.
{
  (*n)++;
  return *n;
}
int add3(int & n) // Reference.
{
  n++;
  return n;
}

// -- Function Calling.
int n = 0;
int q;
q = foo1( n);// n --> n.      q = n + 1.
q = foo2(&n);// n --> n + 1. q = n + 1.
q = foo3( n);// n --> n + 1. q = n + 1.
```

6   Compilation Quirks and Debugging Hints

Always, always, always, write return statements for functions that have a return type. The compiler will not complain if you forget a return statement and will return whatever junk data was in the return register. This may cause your program to have a mysterious segfault

between function calls.

The compiler will not fully build portions of your code, unless there is an execution path from the main function that could possibly call that portion of code. Therefore if you want to see if your code is compiling correctly, then you should integrate it into an actual program, such as a testing

7   Name Spaces

To prevent naming collisions, C++ organizes names into different namespaces. For instance we could define two classes called example, but put them into different namespaces and they would not interfere with each other.

The namespace for the most of the built in classes, such as the data structures is `std`.

To define a namespace for a block of code, for instance with a class declaration, enclose the code with a name space block as follows:

```
namespace exampleNameSpace
{
  class Example
  {
    ...
  }
}
```

To reference the example class you can write one of the following:

```
int main()
{
  exampleNameSpace::Example ex;
  ...
}
```

OR

```
using namespace exampleNameSpace; // This could be seen as name pollution.
int main()
{
  Example ex;
  ...
}
```

If there are name collisions, the compiler will probably yell at you.

8 Data Structures and Operators

8.1 Standard Data Structures

C++ has a standard template library that includes many useful data structures.
Reference: http://www.cplusplus.com/reference/stl/

```
#include <vector>
#include <set>
#include <map>
#include <list>

std::vector<int> v;   // Unbounded Array.
std::set<int> s;      // Binary Search Tree or Set.
std::map<int, int>m;  // Associative Array.
std::list<int>l;      // Linked List.
```

8.2 Operators

8.2.1 Comparators

FIXME : Expand this section.
Here is how to create a custom comparator for use with a map or set.
// Used to impose an ordering for the tuples in the bst.

```
struct ExampleCompare
{
    // Returns true if e1 < e2.
    bool operator()(const Example * e1, const Example * e2) const
    {
      return e1.getNumber < e2.getNumber();
    }
};
```

We specify the less than predicate because C++ can derive all other comparison values from it.

```
Example * a;
Example * b;
a = b --> !((a < b) || (b < a))
a > b --> (b < a)
```

8.2.2 HashValues

Example with both comparators and hash value.
Here is how to specify a global hash value for a given type. You can do this for pointer types, reference types, etc.

```
namespace std
{
  template <>
  struct hash<Example *>
  {
    size_t operator()(const Example * e) const
    {
      // Return hash value.
    }
}
```

8.3   Iterators

FIXME. begin() is inclusive, end() is exclusive.
++, dereferenceing, etc.

9   Useful Stuff

9.1   Math

```
#include <math.h>
```

// Show an example.

9.2   IO and Printing.

```
#include <iostream>
```

// FIXME : Show how to specify working with the cout and ¡¡ operator. // String workings.

9.3   Custom Runtime Errors

```
#include <stdexcept>

void myError()
{
  throw std::runtime_error("ASSERTION FAILED");
}
```

9.4   Templates (And Genarics)

Template specifications must be implemented inside of the header file if your want them to be portable.
// FIXME : Show an example leftist heap.

10 NULL