C++ Quick Guide
v 0.91

Adam Grossman <adamtg@metashadow.com>

Class Definition

```
class <class name>  [: <access level1> <base class name>]
{
    [<access level2>:]
       [explicit] <class name>([<arguments>]) [: <variable name>(<value>)>,];
//constructor

       [friend <class name>|static] <type> <variable name>; // variable
       [friend <class name>|static|virtual] <return type> <method name>([arguments])
// method - inline
       {
              // function body
       }

       [friend <class name>|static|virtual] <return type> <method
               name>([<arguments>])[=0]; // method - prototype

       [friend <class name>;]
       [virtual] ~<class name>(); //destructor

       <type> operator<operator>([argument]);
};

// defining a method if only prototype is declared in class
<class name>::<method name>([<arguments>])
{
       // function body
}
```

class declaration

Field	**Purpose**	**Value/Description**
`<class name>`	name of class	any legal variable name
`<access level1>`	controls access level of base class to anything the uses derived class	**`public`**: access levels from base class remain the same **`protected`**: public members become protected **`private`**: public and protected members become private

class definition

access

Field	**Purpose**	**Value/Description**
`<access level2>`	controls member access for anything using class	There can be as many access level sections in the class as needed, even the multiple sections of the same access level. The deffault level is **`private.`** **`public`**: unrestricted access **`protected`**: only accessible by derived class **`private`**: only accessible by the class itself

constructors

NOTE: A base class's constructor is always called before the derived class's constructor. If the base class needs an

arguments to the constructor, include the parent class constructor in the initialization list
NOTE: If the constructor's single argument is of the same type as the class, and the constructor uses that object to make a copy of it for the newly created object, it's called a copy constructor. If the single argument is of a type other then the class itself, and it is used to convert the argument type to a new object of the constructor's class type, it's called a conversion constructor.

Field	**Purpose**	**Value/Description**
`explicit`	forbids the constructor to be used for an implicit conversion	In order to use the constructor, it must be explicitly called. This avoids inadvertently copy constructors to be called.
`<class name>`	Name of constructor	constructors always are the same name as the class
`<arguments>`	arguments	follows standards C rules, except for one exception. In the method declaration, a default value can be given. The default argument must always start at the first argument, and move right without any non-default arguments between defaulted arguments.
`<variable name>(<value>)>`	Initialization list	Initializes member variables outside of the constructor body. These can only initialize object members, not class members (members declared as **`static`**)

member variable declaration (see "friend" section on the "friend" option):

Field	**Purpose**	**Value/Description**
`static`	makes member variable a class member	Class does not need to instantiated to be accessed. This would be equivalent to creating a global variable within a namespace named <class name>. The same variable is accessed, whether it is referenced from an object or not.
`<type> <variable name>`	variable declaration	follows standard C rules

member method declaration (see "friend" section on the "friend" option):

Field	**Purpose**	**Value/Description**
`static`	makes method a class method	Class does not need to instantiated to be accessed, therefore there can not be a "**`this`**" variable sent for methods. This would be equivalent to creating a global within a namespace named <class name>. The same function is accessed, whether it is referenced from an object or not.
`virtual`	Allows polymorphism	If a pointer of the type base class points to a derived class, the derived class method will be called. In other words, it's not the pointer type which determines the class, but the class the pointer is actually pointing to. If the base class is methods are declared virtual, the derived classes do not to declare it's methods as virtual. As long as the base is declared virtual, those members are considered virtual all the way down the hierarchy.
`<return type> <method name>`	method fingerprint	follows standard C rules
`<arguments>`	arguments	follows standards C rules, except for one exception. In the method declaration, a default value can be given. The default argument must always start at the first argument, and move right without any non-default arguments between defaulted arguments.
`=0`	this makes the function a pure virtual function (must be used with **virtual**)	That means the method is just declared, but not defined. A class with even single pure virtual function can not be instantiated and must have a derived class which defines the every pure virtual

		function

friend

Field	Purpose	Value/Description
friend <class name> [<member>]	allows outside classes access to class	gives the class name <class name> access to the member, regardless of it's access level (if the member is public, it has no effect). If no member is given, the <class name> has access to all of the class's private and protected members

operator overloading
This allows operators to be overloaded to common operators (+,-,[], etc) can be used in a class, hopefully to make the class more intuitive for the user of the class. For binary operators, it's the left side operand which makes the call. For unary operators, the object that makes the call is based on the operator type.

Field	Purpose	Value/Description
<type>	the result of the operator	Based on the operator. if the operator is '=', a bool type would be expected, if the type is '+', the type would be the operator's class to be returned
<operator>	operator being overloaded	Any legal c++ operator, including '+','-','[]', etc. NOTE: Operators like '+' and '+=' are considered different operators. See below for prefix/postfix operator notes.
([argument]**)**	For binary operators, the right side operand is the argument. For unary operators, it's determined by the operators purpose.	Any class or built-in type.

Prefix/Postfix operator overloading
In order to detemine whether the pre- or post- fix operator is used, the following convention is used

```
<type>& operator++();    //prefix
{
 // increment whatever needs to incremented
  return *this;
}

<type> operator++(int); //postfix
{
	<type> temp_obj = *this;
	// increment whatever needs to be incremented in this class

	return temp_obj;
}
```

destructor
NOTE: destructors are called from the derived class first, then the base class

Field	Purpose	Value/Description
virtual	allows polymorphism	behaves just like a virtual method, except the base class destructor will be called. But if the destructor is not

		declared virtual, and the class is destroyed via a pointer to the base class, the derived class's destructor will not be called.
`~<class name>()`	cleans up class	destructors are always the same name as the class and never has any arguments

Method overloading/overriding
Methods can either can either be overloaded (used for methods with common name in the same class) or overridden (used in inheritance). Overloading/overriding is based the methods signature, which is the methods name and arguments. The return value is **not** part of the methods signature.

Overloading	Methods in the same class share the same name, but have different arguments.
Overridding	Method has the same signature has a method in it's base class. The method in the derived class will be called. If the base call and derived class have the same name, but different arguments, it is functionally similar to having overloaded methods in the derived class

iostream

stdin. stdout, stderr
writing to stdout/stderr stream:

```
#include <iostream>

std::[cout|stderr] << <stuff output>| std::endl [<< <stuff to ouput>| std::endl];
```

Field	Purpose	Value/Description
`std::[cout|cerr]`	stream object for stdout and stderr	ostream (no nead to instatiate cout or cerr objects)
`<<`	"pushes" stream to the object	overloaded bit-wise operator
`stuff output`	Data to be outputted	This can be any type of value which supports output, such as built-in types, strings, etc.
`std::endl`	output manipulator	EOL and flush

cin
read in data from stdin

```
#include <iostream>

std::cin >> <variable>;
```

NOTE: **`std::cin`**'s return value evaluates to false in two situations. It read an EOF or the incoming stream contains date that is not a legal value for `variable`.

Field	Purpose	Value/Description
`std::cin`	stream object for stdin	ostream (no nead to instatiate cout or cerr objects)
`>>`	"pushes" stream to the variable	overloaded bit-wise operator
`variable`	variable to hold incoming stream	the data in the stream, as long as it is of a compatible type (integer values being read in for an int variable, etc.)

File I/O
There are three file stream objects

`fstream`	allows reading and writing to a file
`ifstream`	allows just reading from a file
`ofstream`	allows just writing to a file

All three basically behave the same, just and are more restrictive. This allows a stream object to be passed to a method while having compile time checking to make sure the right stream direction object is being passed.

opening a file
NOTE: The arguments in **`File.open`** can also be used for the **`*fstream`** constructor.
NOTE: Only certain modes are valid for certain file stream objects

```
#include <fstream>

[fstream|ifstream|ofstream]  File;

File.open(char *filename, int mode)
```

Field	Purpose	Value/Description
`[stream|ifstream|ofstream]`	stream object	a file stream object
`FILE.open`	method for opening a file	
`char *filename`	file to open	standard C string

`int mode`	modes to open file	Defaults modes:
		`fstream: ios::in | ios::out`
		`ifstream: ios::in`
		`ofstream: ios::out`
		
		See table below for modes

Mode	Description
`ios::in`	Open file to read
`ios::out`	Open file to write
`ios::app`	All the data you write, is put at the end of the file. It calls `ios::out`
`ios::ate`	All the date you write, is put at the end of the file. It does not call `ios::out`
`ios::trunc`	Deletes all previous content in the file. (empties the file)
`ios::binary`	Opens the file in binary mode.

checking for error after opening files:
Two ways:

1. <fstream object>.`open(...)` return values evaluates to false
2. <fstream object>`.fail()` evaluates to false (useful for when file is opened in constructor)

close file:
<fstream object>..`close();`
(do not think there is a need to elaborate)

input methods

```
File >> <value>; //reads one word at a time, does include EOL
File.getch(<char>); //reads one char at time
File.getline(char *string,int length); //reads max at least length chars
```

output method

```
File << <value>; //reads one word at a time, does include EOL
```

```
File.putch(<char>); //write one char at time
File.write(char *string,int length); //reads max at least length chars
```

Casting

All casts are in the form of:
`*_cast<<target_type>>(<source expression>)`

cast type	example	Description/Description

static_cast	`int i;`
	`float f;`
	`f=static_cast<float>(i);`
		Like a C type cast. Does bi-directional pointer casting between base classes and derived class, but does not do safety checks, just makes sure they are compatible (which means incomplete types can be casted)

const_cast	`const char *s1="hello";`
	`char *s2;`
	`s2=const_cast<char *>(s1);`
		Alters the const-ness of an expression

dynamic_cast	`Base *b; // non-polymorphic`
	`Derived *d; // derived from Base`
	
	`d = new Derived();`
	`b=dynamic_cast<Base *>(b);`
		Only used with pointers and references to objects. For non-polymorphic classes, can only case from derived to base class. If the class is polymorphic, the casting can be done in either direction, and verifies that the resulting object will be a complete object. Return NULL on failure.

reinterpret_cast	`ClassA *a;`
	`ClassB *b; // a completely unrelated class`
	
	`b=reinterpret_cast<ClassB *>(a);`
		Blindly casts pointers. There is no safety checking, so unrelated classes can be casted that will cause runtime errors when the pointer is dereferenced.

Const Handling

Basically 'const' applies to whatever is on its immediate left (other than if there is nothing there in which case it applies to whatever is its immediate right).

Example	Description

`const int * Constant2`
`int const * Constant2`
	Declare that **Constant2** is variable pointer to a constant integer

`int * const Constant3`	Declare that **Constant3** is constant pointer to a variable integer
`int const * const Constant4`	Declare that **Constant4** is constant pointer to a constant integer.

`class A`
`{`
`      int func() const;`
`}`
	Adding a const to the end of a method guarantees that no member variables will be altered in the function call.

Templates

Mechanisms for generic types. The entire template (declaration and implementation) must be inside a header file, because it requires a recompilation every time it is used, unlike a standard library

```
template < [(class | typename) identifier [= default][, [(class|typename)
identifier [= default]]*]* | [int variable[=default][,]]*> [function declaration |
class declaration]
```

Field	Purpose	Value/Description
`template`	identifies function or class will use a template	
`(class | typename)`	prepends the template identifier	class or value can be used. they both behave exactly the same way
`identifier`	name which will reference the parameter type	any valid variable name
`= default`	if no type is given, this type will be used	any valid type
`[int variable[=default][,]`	non-type parameter	instead of a type being set at compile, and constant integer value can be set
`[, [(class|typename) identifier[= default type]]*]`	allows multiple parameter types in a single template declaration	

Function Template

Standard function overloading applies, including overloading between template and non-template functions.

Function Template Example

```
template <class T> T function(T x)
{
       x++;
       return x;
}


int i,j;
char x,y;
i=1;
x=1;

j=function(i); // type is inferred by argument
x=function<char>(c); // type is explicitly set
```

Class Template

Class Template Example

```
template <class T1, class T2> class ClassA
{
       T1 x;
       T2 y;

       T1 getX();
       void setY(T2 inY);

};
```

```
template <class T1, class T2> T1  classA<T1,T2>::getX()
       { return x;
       }

template <class T1, class T2> void classA<T1,T2>::setY(T2 inY)
       { y=inY;
       }


// class must be declared with types explicitly set
ClassA<int, float> c;
int x;

x=c.getX();
c.setY(1.02);
```

Specialization
Allows the overriding of the default template implementation for a certain type. In a specialized template, the same methods do not need to be implemented, each can have it's own set of methods, but specialized class will not inherit any undefined methods from the generic template.

```
template <<standard template types identifier> *> class <name of class
being specialized>

template <> class <name of class being specialized> <<type>>
```

Field	**Purpose**	**Value/Description**
`template`	identifies function or class will use a template	
`<<standard template types identifier>>`	same syntax as a standard template identifier	alias for type
`<>`	identifies as an explicit type specialization	
`<name of class being specialized>`	identifies the class being specialized	any valid class name
`<<type>>`	the type that that this template will be specialized	any valid type

Specialization Template Example

```
// standard
template <class T> class ClassA
{
       T x_;
       void set(T x);
};

template <class T> void classA<T>::set(T x)
       { x_=x;
       }

//template to handle pointer
// NOTE: T is not a pointer type.  This syntax just
// allows template specialization for to handle pointers
// if they need to be handled differently
template <class T *> class ClassA
{
       T x_;
       void set(T *x);
};

template <class T> void classA<T>::set(T *x)
       { x_=*x;
       }


//template to handle specific type
template <> class ClassA <char *>
{
       char *x_;
       void set(char *x);
};

void classA<char *>::set(char *x)
       {
          // assuming x_ was allocated somewhere else...
          strcpy(x_,x);
       }
```

Exceptions

Exceptions allow for runtime errors to be handled and control be based to an error handler without any explicit return value checks for every call.

The basic setup for a exception handling is the **`throw-try-catch`** block

```
try
{
  [throw <object> | function with throw <object>]
}
(catch (<[...| type variable]>)
{

})+
```

If there is nothing catch a thrown exception, the program will abort.

Field	Purpose	Value/Description
`try`	marks the beginning of the block which will take a thrown exception and pass it to the **`catch`** blocks	any legal code
`throw` `object`	the object being thrown is what will be caught by the **`catch`** blocks. The **`throw`** can be explicitly in the **`try`** block, or it can be in in any function called within the **`try`** block	any legal object/type
`catch`	one or more blocks to catch the thrown exceptions.	the body of **`catch`** is any legal code
`(``<[...| type variable]>`**`)`**	each **`catch`** has a type it can catch. The **`catch`**es will try each **`catch`** in order, trying to match the **`throw`**n type to the **`catch`** argument ,so ideally, the **`catch`**es should be placed in order of most specific to least specific.	"…" means it will catch any type, but there will not be any argument associated with it. " `type variable`" is a any legal type and any legal variable name.

Rethrowing an exception

If a **`catch`** does not want to handle an exception or wants to have the next level also handle the exception, it can be re**`throw`**n , simply by calling **`throw`** with no argument. The exception will be thrown down to the previous level. In this example, the **`catch`** with the **`int`** argument in main will actually handle the exception.

Rethrowing example

```
int level_2()
{
  std::cout << "Level 2" << std::endl;

  throw 10;

}


int level_1()
{
  std::cout << "Level 1" << std::endl;

  try
    {
      level_2();
    }
  catch (char c)
    {
      std::cout << "L1: Error type char: " << c << std::endl;
    }
  catch (...)
    {
      throw;
    }
}


int main()
{

try
  {
    level_1();
  }
 catch (int i)
   {
      std::cout << "Main: Error type int: " << i << std::endl;
   }
 catch (char c)

   {
      std::cout << "Main: Error type default" <<  std::endl;
   }


}
```

Exception Specifier
Function specifiers specify what type of exceptions (if any) a function, or a function it calls can throw. A function without a function specifier can throw any kind of exception.
NOTE: the exception specifier is not part of the function signature

```
function throw([types,]*)
```

Field	**Purpose**	**Value/Description**
`function`	function prototype or declaration	any valid function
`throw`	signifies an exception specifier	
`[types,]*`	list of 0 or more types of exceptions that can be thrown. If 0 types are listed, then the function will not throw any exceptions.	any valid type

Standard Exceptions
There are some standard exception classes which are used by the C++ Standard Library. They are derived from the **exception** base class.

Exception Class	**Description**
`bad_alloc`	thrown by the **`new`** operator when storage space can not be allocated
`bad_cast`	thrown by dynamic_cast when the casted object would be

	incomplete
`bad_exception`	thrown when a function throws an exception that violates the exception specifier
`bad_typeid`	thrown when **`typeid`** is given a NULL pointer value
`logic_error (and it's derived classes)`	thrown when a logic error occurs which could be detected by reading the code.
`runtime_error (and it's derived classes)`	thrown when a error occurs that could not have been detected by reading the code,.
`ios_base::failure`	base class for **`iostream`** classes

License

Document Changes

Date	Version	Name	Change
07/28/09	0.01	Adam Grossman <adamtg@metashadow.com>	Initial Release
07/30/09	0.02	Adam Grossman <adamtg@metashadow.com>	Added in: Version # in title I/O operator overloading const's casting TBD section
07/30/09	0.03	Adam Grossman <adamtg@metashadow.com>	Added In: Filled in constructor information
08/05/09	0.04	Adam Grossman <adamtg@metashadow.com>	Fixed: errors in cast code samples
10/13/09	0.05	Adam Grossman <adamtg@metashadow.com>	Added in: templates Changed formating of code frames
10/21/09	0.9	Adam Grossman <adamtg@metashadow.com>	Finished: Templates Added in: Exceptions This will be the first official "beta" release
10/21/09	0.91	Adam Grossman <adamtg@metashadow.com>	Added license