The following advice on C++ programming is paraphrased from the work "On to C++" by Patrick Winston, Addison Wesley, 1995

Procedure abstraction hides details of computations inside functions.

Your programs benefit from procedure abstraction by being:

(a) easier to re-use
(b) easier to read
(c) easier to debug
(d) easier to extend
(e) easier to improve
(f) easier to adapt

If you want to make a function with a class-object parameter into a member function, then define the function inside the class definition (as you would an ordinary function) and eliminate the class-object parameter from both the parameter list and from the member variable reference.

If a member function is large, then move the definition outside the class, leaving behind the function prototype:

```
DataType FunctionName (ParameterList);    //prototype
void push (int );        // only need parameter type in prototype
```

indicate the class to which the moved member function belongs as follows:

```
DataType ClassName :: FunctionName (Parameter List)
{    // function body     }
int Stack :: pop () {    // function body   }
```

When you call (invoke) the member function, then use the class member operator (**.** a period), to identify the class object on which the function is to work. The difference would be

```
OrdinaryFunction (ClassArgument, OtherArguments)
reverse (s1, 20);
```

```
ClassArgument.memberFunction (OtherArguments)
s1.reverse (20);
```

Constructors can be used initialize member variables, whenever your program creates a class object.

Each constructor is named for the class in which it is defined. Constructors do not have a return-value data type.

The default constructor has no parameters. You should include a definition of a default constructor in every class that you define.

If you wish to use the default constructor to initialize a class object's member variable values, then use the declaration form:

ClassName VariableName ;

String str; // Not String str ();

Member function in general, and constructors, readers and writers in particular, have complete access to all member variables, public and private.

The member variables and member functions in the public part of a class definition constitute the class' public interface.

The public-private separation helps ensure that you can benefit from data abstraction.

A difference between C and C++

In C++, 0 means false and 1 means true.
(In C, 0 means false and non-zero means true.)

A predicate is a function that returns 0 or 1 (a bool function)

The ! operator means not; The ! operator converts 0 into 1 and any integer other than 0 into a 0.

If you want to force the conversion of a value of one type into the corresponding value of another type, then you must cast the value in the following way:

(type) expression;
(double) 14*5;

If you want to compare two numbers of different types, then you must cast one to match the type of the other.

Input/output streams

If you want to tell C++ that you intend to work with input or output stream, then include the following in your header.

#include <fstream.h>

If you want to open a file

ifstream StreamName (file_specification);
ifstream InputData ("test.data");

For an output file:

ifstream StreamName (file_specification);
ifstream OutputData ("test.results");

Now, to receive data simply replace `cin` by `InputData`
To output results, simply replace `cout` by `OutputData`

Once you have finished with input and output streams then close them with:

StreamName.close ();

Program readability can be improved by replacing integer constants with enumeration constants:

To create a set of enumeration constants, which have consecutive values beginning with 0, then:

enum { FirstConstant, SecondConstant, ... };

If you want your enumeration constants to have specific values then use the pattern:

enum { FirstConst = FirstValue, SecondConst = SecondValue,
...... };

Private, Public and Protected

To limit access to member variables, or member functions, to other member functions defined in the same class, then place them in the private part of the class definition.

To limit access to member variables/functions to other member functions defined in a subclass, then place them in the protected part.

To limit access to member variables/functions, then place them in the public part.

To prevent member variable re-assignment, but yet enable readability of those variables, place them in the private part, but define member-variable readers (access functions) in the public or protected part.

C++ programs normal use call by value parameters. Thus the value of any argument is copied input memory local to the function (reserved for that parameter). This the actual argument is protected from local change.

You can force the C++ compiler to treat the parameter as call-by-reference, by using the & appendage to the formal parameter. Thus the corresponding actual parameter can change through local actions in the function.

DataType FunctionName (..., DataType& ParameterName, ...);
void swap (int&, int&);

To overload (define in a special way) a binary operator, use:

```
DataType operator OperatorSymbol
        ( LeftParamType_and_Name, RightParamType_and_Name )
    {  // code defining the operation }
void BasicStr :: operator= (BasicStr& left, BasicStr right)
    {  // code defining = for use with BasicStr }
```

To overload the output operator, use the pattern:

```
ostream& operator<<
        ( ostream& StreamName, RightParamType_and_Name )
    {   // code defining the operation
        return StreamName;
    }
ostream& operator<< (ostream& s, BasicStr& right)
    {  // whatever you want
        return s;
    }
```

The following material is from the C++FAQ-Lite home page

[16.2] Can I free() pointers allocated with new? Can I delete pointers allocated with malloc()?

No!

It is perfectly legal, moral, and wholesome to use malloc() and delete in the same program, or to use new and free() in the same program. But it is illegal, immoral, and despicable to call free() with a pointer allocated via new, or to call delete on a pointer allocated via malloc().

[16.3] Why should I use new instead of trustworthy old malloc()?

Constructors/Destructors:

Unlike malloc(sizeof(Fred)), new Fred() calls Fred's constructor. Similarly, delete p calls *p's destructor.

Type safety:

malloc() returns a void* which isn't type safe. new Fred() returns a pointer of the right type (a Fred*).

Overridability:

new is an operator that can be overridden by a class, while malloc() is not overridable on a per-class basis.

[16.5] Do I need to check for NULL after p = new Fred()?

No!

If your compiler doesn't support (or if you refuse to use) exceptions, your code might be even more tedious:

```
Fred* p = new Fred();
if (p == NULL) {
  cerr << "Couldn't allocate memory for a Fred" << endl;
   abort();
  }
```

In C++, if the runtime system cannot allocate sizeof(Fred) bytes of memory during p = new Fred(), a bad_alloc exception will be thrown. Unlike malloc(), new never returns NULL!

Therefore you should simply write:

```
Fred* p = new Fred();   // No need to check if p is NULL
```

[16.7] Do I need to check for NULL before delete p?

No!

The C++ language guarantees that delete p will do nothing if p is equal to NULL. Since you might get the test backwards, and since most testing methodologies force you to explicitly test every branch point, you should not put in the redundant if test.

[16.10] How do I allocate / unallocate an array of things?

Use p = new T[n] and delete[] p:

```
Fred* p = new Fred[100];
 // ...
delete[] p;
```

Any time you allocate an array of objects via new (usually with the [n] in the new expression), you must use [] in the delete statement. This syntax is necessary because there is no syntactic difference between a pointer to a thing and a pointer to an array of things (something we inherited from C).

[16.11] What if I forget the [] when deleting array allocated via new T[n]?

All life comes to a catastrophic end.

[16.12] Can I drop the [] when deleting array of some built-in type (char, int, etc)?

No!

Sometimes programmers think that the [] in the delete[] p only exists so the compiler will call the appropriate destructors for all elements in the array. Because of this reasoning, they assume that an array of some built-in type such as char or int can be deleted without the []. That is, they assume the following is valid code:

```
void userCode(int n)
  {
  char* p = new char[n];
   // ...
  delete p;     // <— ERROR! Should be delete[] p !
  }
```

But the above code is wrong, and it can cause a disaster at runtime. In particular, the code that's called for delete p is operator delete(void*), but the code that's called for delete[] p is operator delete[](void*). The default behavior for the latter is to call the former, but users are allowed to replace the latter with a different behavior (in which case they would normally also replace the corresponding new code in operator new[](size_t)). If they replaced the delete[] code so it wasn't compatible with the delete code, and you called the wrong one (i.e., if you said delete p rather than delete[] p), you could end up with a disaster at runtime.

[16.13] After p = new Fred[n], how does the compiler know there are n objects to be destructed during delete[] p?

Short answer: Magic (the system keeps track of this detail).

[26.2] Should I learn C before I learn OO/C++?

Don't bother.

If your ultimate goal is to learn OO/C++ and you don't already know C, reading books or taking courses in C will not only waste your time, but it will teach you a bunch of things that you'll explicitly have to un-learn when you finally get back on track and learn OO/C++ (e.g., malloc(), printf(), unnecessary use of switch statements, error-code exception handling, unnecessary use of #define macros, etc.).

If you want to learn OO/C++, learn OO/C++. Taking time out to learn C will waste your time and confuse you.