A **type-safe** language traps syntax errors at compile-time, diminishing the amount of buggy code that escapes to the client. C++ compilers use type rules to check syntax and generate errors or warnings if any type rule has been violated.

**Encapsulation** is the primary concept of object-oriented programming. It refers to the integration of data and logic within a class' implementation that establishes the crisp interface between the implementation and any client.  Encapsulation maintains high cohesion within a class and low coupling between the class' implementation and any one of its clients.

**Inheritance** - one class inherits the structure of another class

**Polymorphism** - a single interface provides multiple implementations

A well-designed **module** is a highly cohesive unit that couples loosely to other modules. The module addresses one aspect of the programming solution and hides as much detail as practically possible. A compiler translates the module's source code independently of the source code for other modules into its own unit of binary code.

**Preprocessor** - interprets all directives creating a single translation unit for the compiler - (inserts the contents of all #include header files), (substitutes all#define macros)

**Compiler** - compiles each translation unit separately and creates a corresponding binary version.

**Linker** - assembles the various binary units along with the system binaries to create one complete executable binary.

A **reference** is an alias for a variable or object. Object-oriented languages rely on referencing. A reference in a function call passes the variable or object rather than a copy. In other words, a reference is an alternative to the pass by address mechanism available in the C language. Pass-by-reference code is notably more readable than pass-by-address code. To enable referencing, the C++ rules on function declarations are stricter than those of the C language.

*type identifier*(*type*& *identifier*, ... )

***queries*** - also called accessor methods - report the state of the object – *type identifier(args) const;*

***modifiers*** - also called mutator methods - change the state of the object

***special*** - also called manager methods - create, assign and destroy an object

**Constructor** - Complete encapsulation requires a mechanism for initializing data members at creation-time. Without initialization at creation-time, an object's data members contain undefined values until client code calls a modifier that sets that data. Before any modifier call, client code can inadvertently 'break' the object by calling a member function that assumes valid data.

**Destructor** - Complete encapsulation also requires a mechanism for tidying up at the end of an object's lifetime. An object with dynamically allocated memory needs to deallocate that memory before going out of scope. An object that has written data to a file needs to flush the file's buffer and close the file before going out of scope

The keyword **this** returns the address of the current object. That is, **this** holds the address of the region of memory that contains all of the data stored in the instance variables of current object. **\*this** refers to the current object itself; that is, to the complete set of its instance variables.

We classify operators by the number of operands that they take:

* **unary** - one operand - post-fix increment/decrement, pre-fix increment/decrement, pre-fix plus, pre-fix minus
* *return\_type* operator *symbol*()
* **binary** - two operand - assignment, compound assignment, arithmetic, relational, logical
* *return\_type* operator *symbol* (*type [identifier]*)
* **ternary** - three operands - conditional operator → () ? :

Type conversion operators define implicit conversions to different types, including fundamental types

*operator [type]() const;*

We can use **temporary objects** to access validation logic localized within one constructor. Note the temporary object assignments to the current object (\*this) in the one-argument and three-argument constructors below

In designing a class with a resource, we expect the resource associated with one object to be independent of the resource associated with another object. That is, if we change the resource data in one object, we expect the resource data in the other object to remain unchanged. In copying and assigning objects we ensure resource independence through deep copying and deep assigning. Deep copying and deep assigning involve copying the resource data. Shallow copying and assigning involve copying the instance variables only and are only appropriate for non-resource instance variables.

In each deep copy, we allocate memory for the underlying resource and copy the contents of the source resource into the destination memory. We shallow copy the instance variables that are NOT resource instance variables. For example, in our Student class, we shallow copy the student number and number of grades, but not the address stored in the grade pointer.

Two special member functions manage allocations and deallocations associated with deep copying and deep copy assigning:

* the copy constructor
* the copy assignment operator

If we do not declare a copy constructor, the compiler inserts code that implements a shallow copy. If we do not declare a copy assignment operator, the compiler inserts code that implements a shallow assignment

*Type*(const *Type*&) [ = delete]; //copy constructor

*Type*& operator=(const *Type*&) [ = delete]; //copy operator

Certain class designs require prohibiting client code from copying or copy assigning any instance of a class. To prohibit copying and/or copy assigning, we declare the copy constructor and/or the copy assignment operator as deleted members of our class: