**Vu Pham 129908174**

**A *namespace*** is a scope for the entities that it encloses.  Scoping rules avoid identifier conflicts across different namespaces.

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.

**a header file** - defines the class and declares the function prototypes // **an implementation file** - defines the functions and contains all of the logic

**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

The memory that the operating system allocates for the application at load time is called ***static memory*.**  Static memory includes the memory allocated for program instructions and program data.  The compiler determines the amount of static memory that each translation unit requires. The application's variables and objects share static memory amongst themselves. When a variable or object goes out of scope its memory becomes available for newly defined variables or objects.

The memory that an application obtains from the operating system during execution is called ***dynamic memory*.**  Dynamic memory is distinct from the static memory.  While the operating system allocates static memory for an application at load time, the system reserves dynamic memory, allocates it and deallocates it at run-time. **The lifetime** of any dynamically allocated memory ends when the pointer holding its address goes out of scope.  The application must explicitly deallocate the allocated region of dynamic memory within this scope.  If the application neglects to deallocate the allocated region, that memory becomes inaccessible and irrecoverable until the application returns control to the operating system. Unlike variables and objects allocated in static memory, those in dynamic memory do not automatically go of out scope at the closing brace of the code block within which they were defined.  We must manage their deallocation explicitly ourselves.

**The nullptr** keyword identifies the address pointed to as the null address.  This keyword is an implementation constant.

**Memory leaks** are one of the most important bugs in object-oriented programming.  A memory leak occurs if an application loses the address of dynamically allocated memory before that memory has been deallocated.  This may occur if : 1// [Memory leaks are difficult to find because they often do not halt execution immediately.  We might only become aware of their existence indirectly through subsequently incorrect results or progressively slower execution.]

**1.the** pointer to dynamic memory goes out of scope before the application deallocates that memory // **the pointer** to dynamic memory changes its value before the application deallocates the memory starting at the address stored in that pointer

**Insufficient Memory** On small platforms where memory is severely limited, a realistic possibility exists that the operating system might not be able to provide the amount of dynamic memory requested.  If the operating system cannot dynamically allocate the requested memory, the application may throw an exception and stop executing

**member functions** of a class provide the communication links between client code and objects of the class.  Client code calls the member functions to access an object's data and possibly to change that data

We can validate information incoming from client code before storing it in an object.  If the data is invalid, we can reject it and store default values that identify the object's state as an ***empty state***

The special member function that any object invokes at creation-time is called its class**' *constructor***.  We use the default constructor to execute any preliminary logic and set the object to an empty state. **allocates memory** for each instance variable in the order listed in the class definition. **executes** the logic, if any, within the constructor's definition

The special member function that every object invokes before going out of scope is called its class'***destructor*.**  We code all of the terminal logic in this special member function.  **execute the** logic of the object's destructor. **deallocate** memory for each instance variable in opposite order to that listed in the class definition //destruct: is called automatically, cannot be overloaded, should not be called explicitly.

Char \* c =nullptr;

C= new char[size];

Strcjhghjjh

Delete[] c;

C = nullptr;

In the C++ language, the keyword **operator** identifies an overloaded operation **Student& operator=(const Student&); Student& Student::operator+=(float g) { if (no != 0 && ng < NG && g >= 0.f && g <= 100.f) grade[ng++] = g; return \*this; }**

**Student& operator++(); Student& Student::operator++() { for (int i = 0; i < ng; i++) if (grade[i] < 99.0f) grade[i] += 1.f; return \*this; }**  
**Student operator++(int);**

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
* binary - two operand - assignment, compound assignment, arithmetic, relational, logical
* ternary - three operands - conditional operator

We overload operators in either of two ways, as:

* member operators - part of the class definition
* helper operators - supporting, but outside the class definition

The definition of a copy constructor contains logic to

1. perform a shallow copy on all non-resource instance variables
2. allocate memory for each new resource
3. copy data from the source resource to the newly created resource

**Student::Student(const Student& src) {**

**// shallow copies**

**no = src.no;**

**ng = src.ng;**

**// allocate dynamic memory for grades**

**if (src.grade != nullptr) {**

**grade = new float[ng];**

**// copy data from the source resource**

**// to the newly allocated resource**

**for (int i = 0; i < ng; i++)**

**grade[i] = src.grade[i];**

**} else**

**{ grade = nullptr; } }**

**Student& operator=(const Student&);**

**Student& Student::operator=(const Student& source) { // check for self-assignment if (this != &source) { // shallow copy non-resource variables no = source.no; ng = source.ng; // deallocate previously allocated dynamic memory delete [] grade; // allocate new dynamic memory, if needed if (source.grade != nullptr) { grade = new float[ng]; // copy the resource data for (int i = 0; i < ng; i++) grade[i] = source.grade[i]; } else { grade = nullptr; } } return \*this; }**