C/C++ Tutorial for Computer Graphics

Part I

The tutorial explains the C/C++ basics that will be used in Computer Graphics.

Outline

1. Integrated Development Environment (IDE)

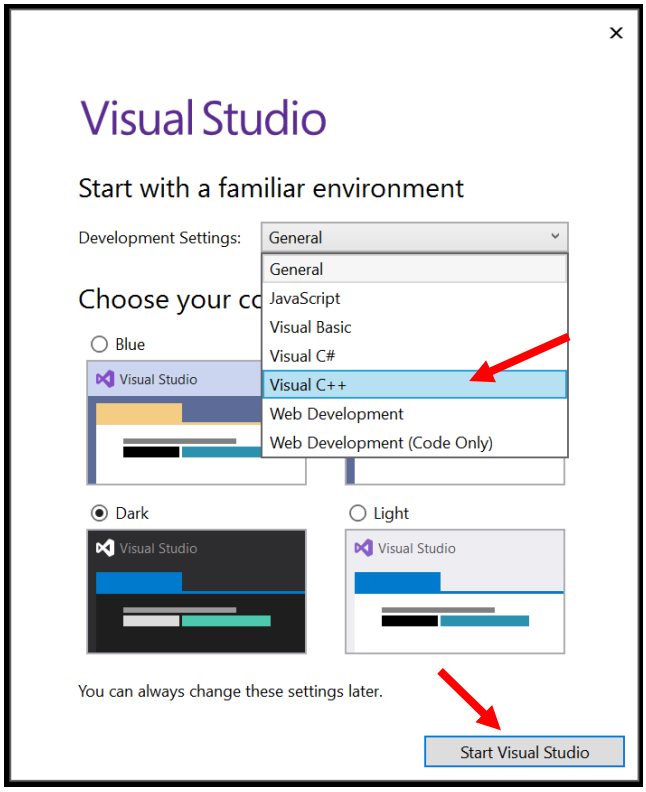
While we are not going into detail about the compiler here, you may find it helpful to get a good understanding of the concept before using the integrated development environment (IDE). You can get this information in any C/C++ book. We will introduce the basic operations of Microsoft IDE, Visual Studio 2019.

Visual Studio 2019 is a tool from Microsoft that integrates a development interface and the toolkits needed to compile a variety of programming languages, e.g. C/C++, C#, Visual Basic and so on. It is much similar to NetBeans which you have used in Object Oriented Programming (e.g. CSE 105). You can download it from: https://www.visualstudio.com/zh-hans/vs/older-downloads/

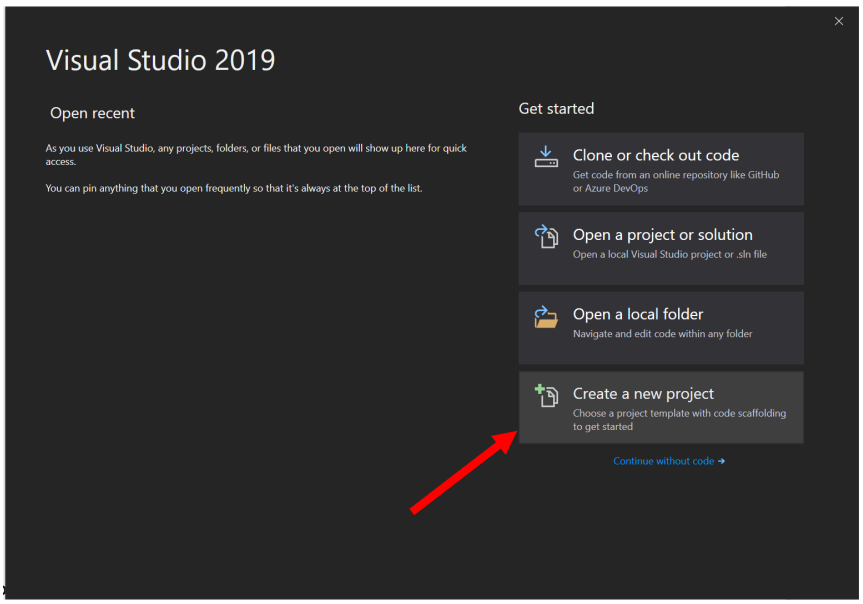
Installation

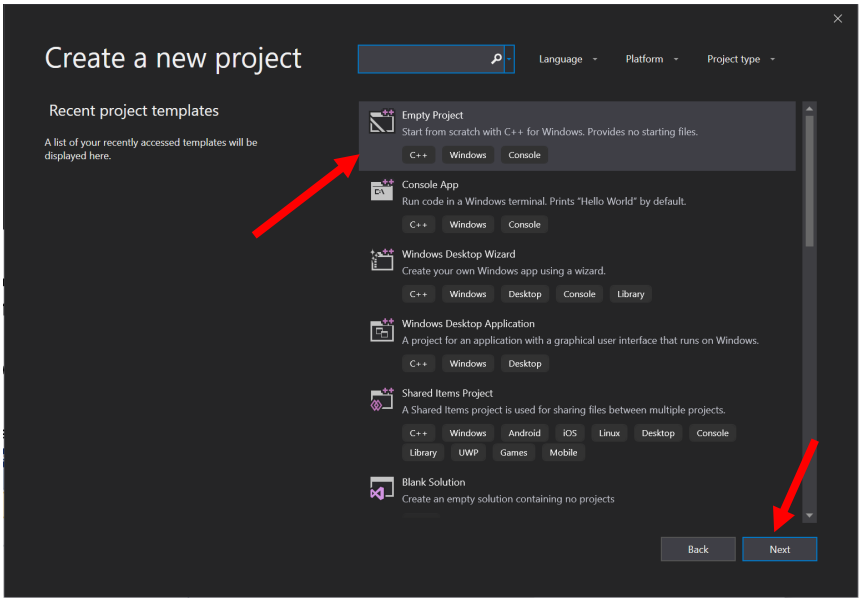
If you are using Windows 7 or above operation system, you can run the executable file which you have download from the website directly. If it is an ISO, open the ISO file and run the executable file within it. Just follow the default setting given by the installer.

If you open the Visual Studio 2019 which you have just installed, you will see the default environment selection. Select Visual C++ Development Setting and click Start Visual Studio button.

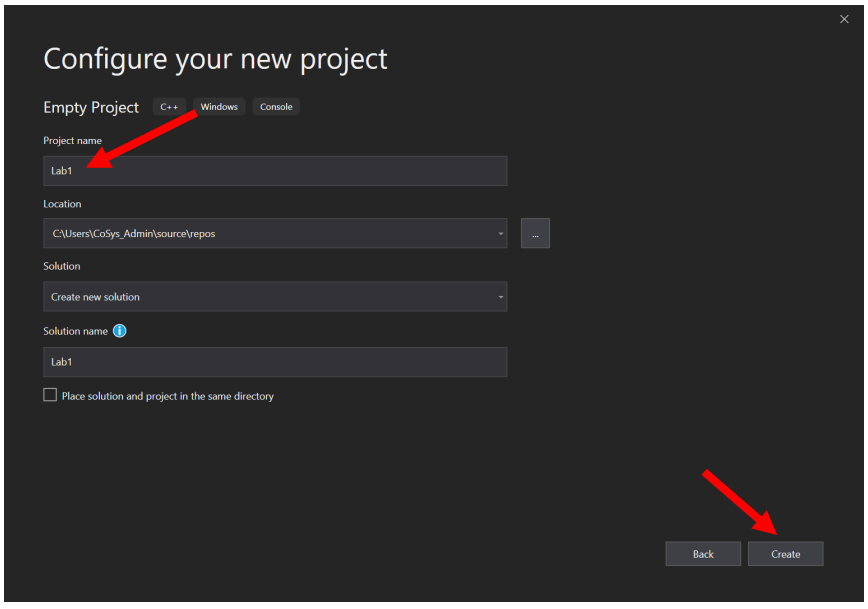


1. Then click 'Create a new project, and you will see the following window. Choose Empty Project and click Next.

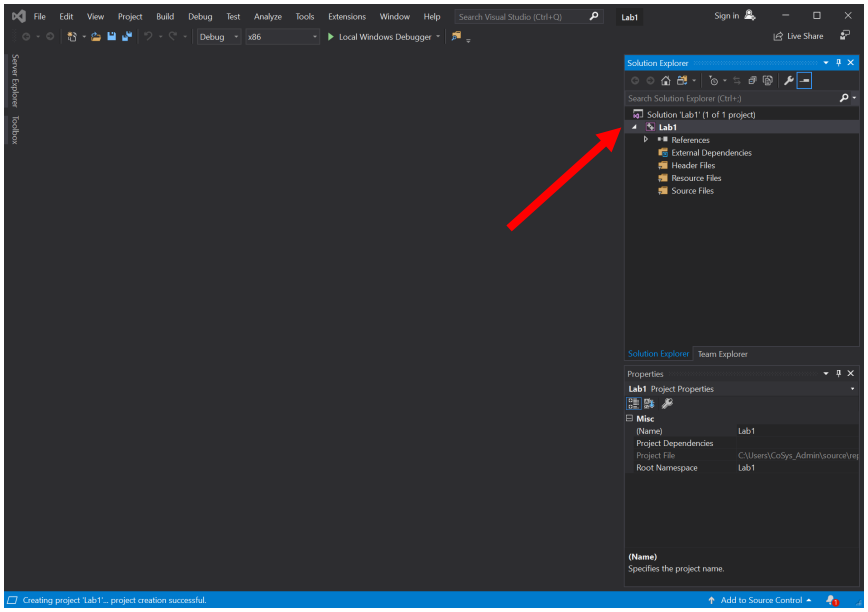




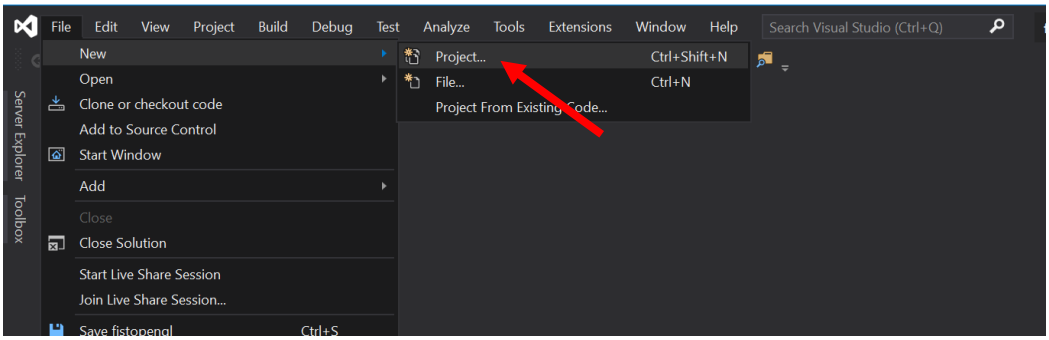
2. Input a name for the project such as that in the following picture, ‘Lab1’. Then “Create”.



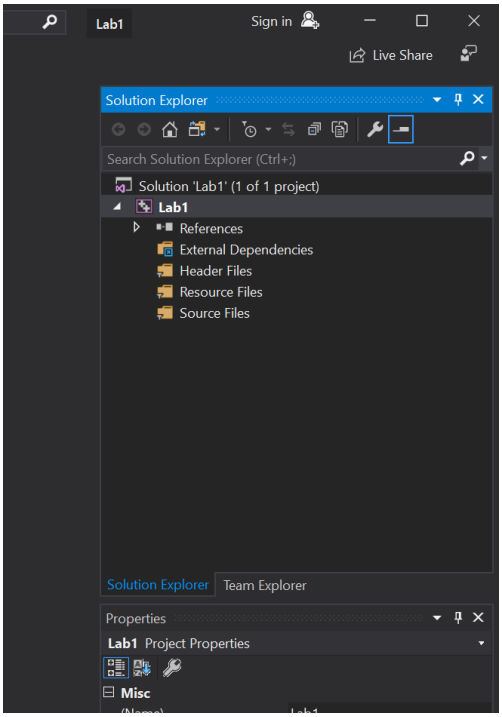
3. Now you have created a new project.



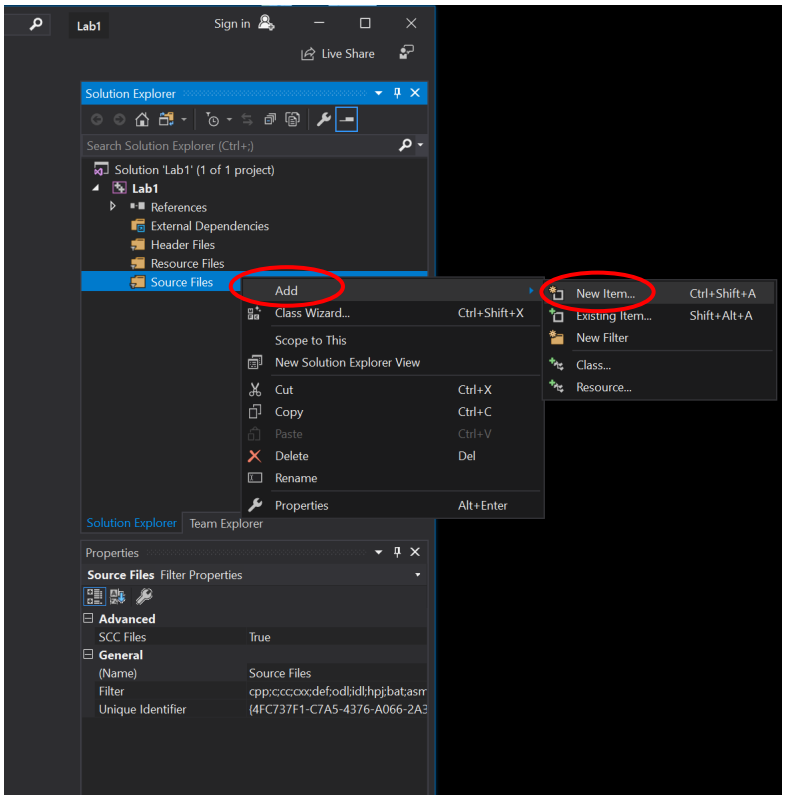
Note: if you want to create another project, choose *File -> New->Project*



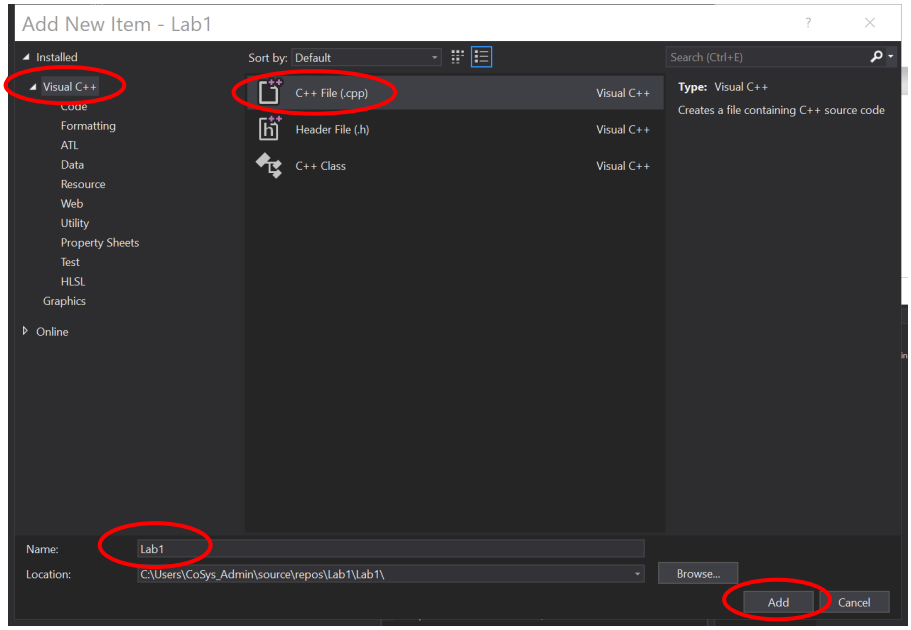
4. Now you have an empty project. You need to add a file to it. For that:



On the *Solution Explorer* at left, look for *Source Files* folder under your application. Right-click -> Add -> New Item.



Here, add a new C++ file:



You can give this file any name with a *.cpp* extension, i.e. *Lab1.cpp*. After clicking *Add*, the main windows will display a code editor to edit the C++ file. Just write the sample code in your lab work tutorial in it. Then to compile and run this application simply press Ctrl+F5. You can edit this file as much as needed and trigger a new compile and run every time when ready.

2. Structure of Program

Let's write the first C++ program.

```
// Sample code
#include <iostream>

int main()
{
        std::cout<<"Hello CPT205!";
}
```

Line 1: //The tutorial sample code in C++

It is same with JAVA. In C++, the slash-slash (*//*) signs indicate that the rest of the line is a comment inserted by the programmer but which has no effect on the behavior of the program.

Line2: #include <iostream>

It is similar to JAVA. If you want to use a Java library, you need to import it at first. In C++, if you want to use a C++ library, you need include its head file at first. In this case, the directive *#include <iostream>*, known as header iostream, that allows to perform standard input and output operations, such as writing the output of this program (*Hello World !*) to the screen.

Line3: A blank line

Blank lines have no effect on a program. They can improve readability of the code.

Line4: int main()

It is similar to JAVA. The function named main is a special function in all C++ programs; it is the function called when the program is run. The execution of all C++ programs begins with the main function, regardless of where the function is actually located within the code. This line initiates the declaration of the "main" function. The definition is introduced with a succession of a type (*int*), a name (*main*) and a pair of parentheses (*()*), optionally including parameters.

Lines 5 and 7: { and }

It is same with JAVA. The definition of main function should write between the open brace (*{*) at line 5 indicates and the closing brace (*}*) at line 7. All other functions use braces to indicate the beginning and end of their definitions.

Line 6: std::cout << "Hello CPT205!";

This line is a C++ statement. A statement is an expression that can actually produce some effect. It is the meat of a program, specifying its actual behavior. Statements are executed in the same order that they appear within a function's body.

This statement has three parts: First, *std::cout*, which identifies the standard character output device (usually, this is the computer screen). Second, the insertion operator (<<), which indicates that what follows is inserted into *std::cout*. Finally, a sentence within quotes (*"Hello CPT205!"*), is the content inserted into the standard output. If you do not want to add *std::* before the *cout*, you can add *using namespace std;* in line 3.

Notice that the statement ends with a semicolon (*;*). This character marks the end of the statement, just as the period ends a sentence in English. All C++ statements must end with a semicolon character. One of the most common syntax errors in C++ is forgetting to end a statement with a semicolon.

3. Constants, Variables and Arrays

In C++, the declarations of the constant and variable are similar to JAVA.

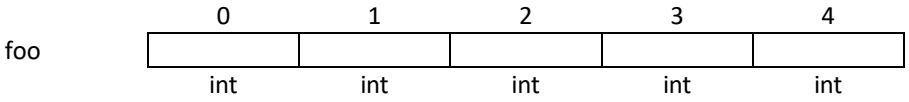
```
// Sample code
#include <iostream>
using namespace std;

const double pi = 3.14159;

int main ()
{
     double r;  // radius
     double circle;  // circumference of circle
     cout << "Please enter radius: ";
     cin >> r;

     circle = 2 * pi * r;
     cout << "Circumference of circle is: " << circle <<endl;
}
```

Line 5 is a definition of constant pi. Please pay attention to the key work "const". Line 9 and line 10 are definition of variables. Line 12 is mathematics expression which can calculate the value of variable circle. In line 13, "*endl*" is a manipulator in C++. It means end of line.

For array in C++, it is different from JAVA. A typical declaration for an array in C++ is:
type name [elements];

where type is a valid *type* (such as *int, float* ...), *name* is a valid identifier and the *elements* field (which is always enclosed in square brackets *[]*), specifies the length of the array in terms of the number of elements.

For example, an array containing 5 integer values of type *int* called *foo* could be represented as:



Therefore, the *foo* array, with five elements of type *int*, can be declared as:

int foo [5];

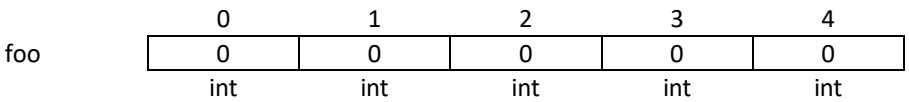
NOTE: The *elements* field within square brackets *[]*, representing the number of elements in the array, must be a *constant expression*, since arrays are blocks of static memory whose size must be determined at compile time, before the program runs.

By default, regular arrays of *local scope* (for example, those declared within a function) are left uninitialized. This means that none of its elements are set to any particular value; their contents are undetermined at the point the array is declared.

The initializer can even have no values, just the braces:

int foo [5] = { };

This creates an array of five int values, each initialized with a value of zero:



The elements in an array can also be explicitly initialized to specific values. For example:

int foo [5] = { 10, 20, 30, 40, 50 };

This statement defines an array that can be represented like this:

	0	1	2	3	4
foo	10	20	30	40	50
	int	int	int	int	int

The values of any of the elements in an array can be accessed just like the value of a regular variable of the same type. The syntax is:

name[index]

Following the previous examples in which foo had 5 elements and each of those elements was of type *int*, the name which can be used to refer to each element is the following:

	foo[0]	foo[1]	foo[2]	foo[3]	foo[4]
foo	10	20	30	40	50
	int	int	int	int	int

For example, the following statement stores the value 50 in the fifth element of foo:

foo[4] = 50;

and, for example, the following copies the value of the third element of foo to a variable called x:

x = foo[2];

NOTE: In C++, the first element of foo is *foo[0]*. The last element of foo is *foo[n-1]*. And n is the number of elements in foo array.

Multidimensional arrays can be described as "arrays of arrays". For example, a two-dimension array can be imagined as a two-dimensional table made of elements, all of them of the same uniform data type.

		0	1	2	3	4
	0	foo[0][0]	foo[0][1]	foo[0][2]	foo[0][3]	foo[0][4]
foo	1	foo[1][0]	foo[1][1]	foo[1][2]	foo[1][3]	foo[1][4]
	2	foo[2][0]	foo[2][1]	foo[2][2]	foo[2][3]	foo[2][4]

Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as needed. Although be careful: the amount of memory needed for an array increases exponentially with each dimension. For example:

char century [100] [365] [24] [60] [60];

declares an array with an element of type *char* for each second in a century. This amounts to more than 3 billion *char*! So this declaration would consume more than 3 gigabytes of memory.

At the end, multidimensional arrays are just an abstraction for programmers, since the same results can be achieved with a simple array, by multiplying its indices:

```
int foo [3] [4];  // is equivalent to int
foo [12];         // (3*4 = 12)
```

4. Data Structure

For a *data structure* in C++, it is similar to the class definition in JAVA. It is a set of data elements of the same type or different type under one name.

1. The declaration of the data structure and the data structure variables:

(1) A typical declaration for a data structure is to declare the structure type first and to declare the structure variables:

```
struct type_name
{
        member_type1 member_name1;
        member_type2 member_name2;
};
type_name object_name;
```

where *type_name* is a valid identifier for the structure type and *object_name* is a valid identifier of object having this structure type. There is a member list in braces {} and we can declare the member variables just like declaring regular variables, such as *int a*.

For example, it declares a structure called *Student* and defines it consisting of two data members of different type: *name* and *IDnum*, as follows:

```
struct Student
{
        char name[20];
        int IDnum;
};
Student Stu1;
Student Stu2;
```

Follow the declaration of this structure (*Student*), we can use it to declare two variables of this type: *Stu1* and *Stu2*. If we need to access the member of structure variable, we can use a dot (.) operator. For example:

```
Stu1.IDnum = 1234567;
Stu2.IDnum = 1234568;
```

Other methods:

(2) The structure variables are declared at the moment the data structure type is declared

```
struct Student
{
        char name[20];
        int IDnum;
} Stu1, Stu2;
```

(3) The variables of structure type are directly declared

```
struct
{
        char name[20];
        int IDnum;
} Stu1, Stu2;
```

The third method is not common and we usually use the first one.

2. Initialization of the structure variables:

(1) To initialize variable values when declaring variables (this method is more commonly used)

```
struct Student
{
        char name[20];
        int IDnum;
};
Student Stu1= {values};
Student Stu2= {values};
```

(2) To specify the initial values for structure variables when declaring the structure

```
struct Student
{
        char name[20];
        int IDnum;
} Stu1={values}, Stu2={values};
```

An example as follows:

```
// Sample code
#include <iostream>
using namespace std;

// Declare the structure variable Stu1 and assign the value
struct Student
{
     char name[20];
     int IDnum;
} Stu1 = {"Li Ming", 123456};

void main()
{
     // Declare the structure variable Stu2 and assign the value
     Student Stu2 = {"Liu Hong", 654321};

     // Access to the members in the structure variable Stu1 through the member operator
     cout <<"First student is:"<<Stu1.name<<"\n ID number is:"<< Stu1.IDnum << endl;

     // Access to the members in the structure variable Stu2 through the member operator
     cout <<"Second student is:"<<Stu2.name << "\n ID number is:"<< Stu2.IDnum << endl;
}
```

3. Declaration of an array of structure type

A structure variable can store a set of data (such as a student's number, name, grade, etc.). If there are 10 students who need to participate in the data, it is obvious that the array should be used, which is the array of structure. Declaring an array of structure is the same to declare a structure variable. For example:

```
struct Student
{
```

```
        char name[20];
        int IDnum;
};
Student Stu[2];
```

For the initialization of an array of structure, it is same to the initialization of a structure variable. For example:

```
struct Student
{
        char name[20];
        int IDnum;
};
Student Stu [2] = {
        {“Li Ming”, 123456},
        {“Liu Hong”, 654321}
};
```

When we define an array *Stu* [ ], we can also specify no number to elements, that is: Student Stu[ ] = {{…},{…},{…}};

An example as follows:

```
// Sample code
#include <iostream>
using namespace std;

// Declare a structure
struct Student
{
    char name[20];
    int IDnum;
};

void main()
{
    // Declare an array of structure and assign the value
    Student Stu[2] = {
          {"Li Ming", 123456},
          {"Liu Hong", 654321}
    };

    // Access to the members in the array of structure Stu[0] through the member operator
    cout <<"First student is:"<<Stu[0].name<<"\n ID number is:"<< Stu[0].IDnum << endl;

    // Access to the members in the array of structure Stu[1] through the member operator
    cout <<"Second student is:"<<Stu[1].name << "\n ID number is:"<< Stu[1].IDnum << endl;
}
```