

Project 2

Name:Dazhi Li

RUID:dl939

1. C++ Scope Rules in applying to code blocks, functions, global namespace, function-prototype

C++ scope rules mean there is global and local variable in different blocks. Here is an example:

```
#include<iostream>

using namespace std;//Global namespace
int a=10;//Global variable

int SetValue();//Function Prototype

int main(void)
{//code block begins
    int b=100;//local variable, only accessible in main function
    cout << "This is a:"<< a << endl;
    cout << "This is b:" << b << endl;
    SetValue();
    cout << "This is a:" << a << endl;
    return 0;
}//code block ends

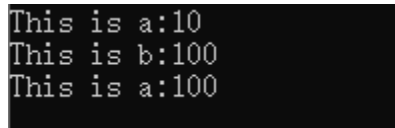
//b = 10; If you want to use variable b out of main function, error happened

int SetValue()
{//code block begins
    a = 100; //Global value could be accessed through the whole code
    return 0;
}//code block ends
```

In this example we can find function SetValue(). This function will set the global value a to 100 to prove that global value could be accessed through anywhere in the code.

Global namespace is used for declaration for using some tools in a library. In this example we are using global namespace std. This will announce that we are using C++ standard library to our compiler. So we can use cout instead of std::cout in our code.

As for function prototype, I just give an example which is SetValue() function. At the beginning of this code I just give a declaration for it and then implement it later. Function prototype is used for creating library interface or function declaration.



```
This is a:10
This is b:100
This is a:100
```

Fig 1 Running result

2. function-call stack, stack frames, automatic variables and stack frames, stack overflow
function-call stack and stack frames are too abstract for me to write a code to show what they are. Here I will just explain what they mean to me. In our code above, we call a function named SetValue(). Also, we have another function which is our main function. When SetValue() is called from program, C++ compiler will create a stack frame to store the function information. That is where stack frame was used in function call. And the function call stack will be running into the SetValue function and loading those parameters in it. At the end of the function, when it exits the function call stack will point to the next row of code after the function. That is how function-call stack works in C++.

As for the automatic variables and stack overflow I will give an example code on what they are.

```
int main()
{
    int a;//Automatic variable a was created
    int B[3] = { 0,1,2 };
    //cout << "a is:" << a << endl; since a is not given a value, we cannot visit
    cout << "The address of a is:" << &a << endl;
    cout << "B[0] is:" << B[0] << endl;
    cout << "B[1] is:" << B[1] << endl;
    cout << "B[2] is:" << B[2] << endl;
    //cout << "B[3] is:" << B[3] << endl; This is a typical stack overflow
    return 0;
}
```

In this simple example code, we will find an automatic variable "a". This variable will not be given any value. But program will create a memory space for it. But we cannot visit it since it has no value. But we could visit the memory address the program allocated. This is a typical automatic variable. In C++, the constructor of automatic variables is called when the execution reaches the place of declaration. The destructor is called when it reaches the end of the given program block (program blocks are surrounded by curly brackets).

Then, B is an int type array we created. The size is three and we set value to it. We can visit all the three number in this array since it only has three numbers in it. But if we want to visit the fourth number in it, we will be thrown a compiler error which is stack overflow. Stack overflow is a very typical problem in our coding. The stack only allocates three memory space for it but we want to find the fourth one, which is impossible.

```
The adress of a is:008FFEC0
B[0] is:0
B[1] is:1
B[2] is:2
```

Fig 2 Running result

3. function default parameters

Example code:

```
int PrintPoint(int a=0, int b=0)//Default parameters when function called
{
    cout << "The point of yours is (" << a << ", " << b << ")" << endl;
    return 0;
}

int main(void)
{
    PrintPoint();
    PrintPoint(1);
    PrintPoint(3, 3);
}
```

Usually when we setup a function with some arguments in it, we need user to input some parameters for them. However, we can setup default function parameters in a function like the PrintPoint() function above. When users do not input anything, this function will print out the origin. However, if the user input something it will just print our user's customized point position. This typology could also be used in class member functions and constructors.

```
The point of yours is (0,0)
The point of yours is (1,0)
The point of yours is (3,3)
```

Fig 3 Running result

4. inline functions

Inline function is used for increasing the function call efficient. Usually we will make some small function to be inline function to save the stack memory. It will save the time for pushing in stack and popping out of stack. And before you call an inline function you must declare and implement it first. Also loop and switch operation is not allowed to show up in inline function since it is just kind of recommendation for the compiler to make the function more efficient.

```
inline int Max(int x, int y)//Inline function
{
    return (x > y) ? x : y;
}

int main()
{
```

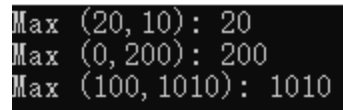
```

    cout << "Max (20,10): " << Max(20, 10) << endl;
    cout << "Max (0,200): " << Max(0, 200) << endl;
    cout << "Max (100,1010): " << Max(100, 1010) << endl;

    return 0;
}

```

In this code example we set an inline function Max which will find the maximum value between two integers. Inline function will increase the speed of function calling as I mentioned above. So, when we call this function three times it will just running without three times of pushing in stack and popping out of stack.



```

Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010

```

Fig4 Running result

5. function arguments pass-by-value, pass-by-reference, and how-to unary scope operator

Function passed by value is what we usually do in using a function. In this case, we input our parameters as arguments of the function.

Function pass by reference will use the reference of the parameter as arguments of a function. See the code below:

```

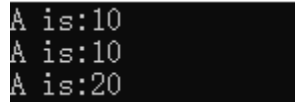
int pass_by_value(int a)
{
    a = 20;
    return 0;
}

int pass_by_reference(int& a)
{
    a = 20;
    return 0;
}

int main()
{
    int a = 10;
    cout << "A is:" << a << endl;
    pass_by_value(a);
    cout << "A is:" << a << endl;
    pass_by_reference(a);
    cout << "A is:" << a << endl;
    return 0;
}

```

In this case, function pass_by_value does not change the value of variable a in main function as it just change the local value in it. But pass_by_reference will change the value of a because in this function we are doing operation on the address of a.



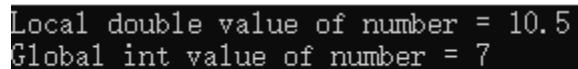
```
A is:10
A is:10
A is:20
```

图 5-1 Running result

As for the unary scope operator, C++ provides the unary scope resolution operator (::) to access a global variable when a local variable of the same name is in scope.

Code example:

```
int number = 7; // global variable named number
int main()
{
    double number = 10.5; // local variable named number
    cout << "Local double value of number = " << number << endl;
    cout << "Global int value of number = " << ::number << endl;
    return 0; // indicates successful termination
} // end main
```



```
Local double value of number = 10.5
Global int value of number = 7
```

Fig 5-2 Running result

6. function overloading

In C++, it can have function in the same name. But we should make the argument type different from each other so that we could achieve function overloading.

Code example:

```
int print(int a)
{
    cout << "This is a int type " << a << endl;
    return 0;
}

int print(float a)
{
    cout << "This is a float type " << a << endl;
    return 0;
}

int main()
{
    print(int(10));
    print(float(0.1));
    return 0;
}
```

```
This is a int type 10
This is a float type 0.1
```

Fig 6 Running result

7. function templates

```
template <class myType> //Create a function template
myType GetMax(myType a, myType b)
{
    return (a > b ? a : b);
}
int main()
{
    int a = 10, b = 5, c;
    float d;
    c = GetMax(a, b);
    cout<< "The larger number is:" << c << endl;
    //d = GetMax(10.5, b); Compiler error if you use two different types as templates
    return 0;
}
```

In this example, we assume a template myType , this type could be any datatype when it is used. But we set a function GetMax() which will use myType as arguments and return type. This will ask the user to input the same datatype. If the datatype is different, compiler will throw us an error.

```
The larger number is:10
```

Fig 7 Running result

8. Recursion, Recursion vs. Iteration

Recursion is a common programming method which is broadly used. Recursion is achieved by function call function itself to form an iteration. Here is an example code on how to achieve Fibonacci sequence:

```
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n - 1) + fib(n - 2);
}

int main()
{
    int n = 9;
    cout << n << " sequence fibonacci is: " << fib(n) << endl;
    return 0;
}
```

```
9 sequence fibonacci is: 34
```

Fig 8 Running result

Comparing to recursion method, iteration is also another method which is efficient to achieve loops. Iteration is easier for programmer to code, but recursive method is better readable. Recursion also will be easy to fall in infinite iteration if programming wrong.

9. built-in array, array class template, vector class template

Built-in array could be directly used by C++ coders. User just need to assign a datatype, array name and array length to create it.

As for array class template and vector class template, here is the code example:

```
template< typename T > class array {
private:
    int size;
    T* myarray;
public:
    // constructor with user pre-defined size
    array(int s) {
        size = s;
        myarray = new T[size];
    }
    // calss array member function to set element of myarray
    // with type T values
    void setArray(int elem, T val) {
        myarray[elem] = val;
    }

    // for loop to display all elements of an array
    void getArray() {
        for (int j = 0; j < size; j++) {
            // typeid will retriev a type for each value
            cout << setw(7) << j << setw(13) << myarray[j]
                 << " type: " << typeid(myarray[j]).name() << endl;
        }
        cout << "-----" << endl;
    }
};
```

This is an example on how to set a template class array. As we do not know the array datatype and length at the beginning, but we can use template to reuse it in different datatypes. And the template vector array class is just like the same.

10. multidimensional array, how to

Multidimensional array could be achieved by built-in array or built-in vector. This is widely used in high dimensional data store and manipulation. The example code is shown below:

```
int main(void)
{
```

```

string Student_info[2][10];
Student_info[0][0] = "Eric";
Student_info[1][0] = "boy";
Student_info[0][1] = "Alice";
Student_info[1][1] = "girl";
for (int i = 0; i < 2; i++)
{
    for (int j = 0; j < 2; j++)
    {
        if (j == 0)
        {
            cout << "Student: " << Student_info[j][i] << " ";
        }
        else if (j == 1)
        {
            cout << "Sex: " << Student_info[j][i] << endl;
        }
    }
}
}

```

In this case we use two-dimensional array to store the student information about name and sex. If we want to store more information like their heights, we could use three or more dimensions.

```

Student: Eric Sex: boy
Student: Alice Sex: girl

```

Fig 10 Running result

11. Pointers, using them appropriately, correctly, responsibly

Pointer is commonly used in C++ and it will point to an object, whose datatype could be class, int, double and even template type. Pointer will store the address of the object which it is pointing. Here is the code example:

```

int main(void)
{
    int i = 10;
    int* i_pointer; //Create a pointer
    i_pointer = &i; //Set the pointer pointing to variable i
    cout << "i is: " << i << endl;
    cout << "The address of i is: " << &i << endl;
    cout << "i_pointer is: " << i_pointer << endl;
    cout << "What i_pointer pointing is: " << *i_pointer << endl;
    return 0;
}

```

In this case we set a variable i and a pointer named i_pointer. i_pointer is set to point at i. From the running result you will find what a pointer is.


```
i is: 10
The address of i is: 0083FC94
i_pointer is: 0083FC94
What i_pointer pointing is: 10
```

Fig 11 Running result

12. *, & at declaration

“*” has two different meaning: The first one is setting a pointer, which will be used as datatype* pointer_name. The second usage is visiting the object which it points at.

“&” also has two different meaning: The first meaning is visiting the address of one object. The other meaning is setting a reference to an object.

Example code:

```
int main(void)
{
    double object;
    double* object_pointer; //First usage of "*": Setting pointer
    object_pointer = &object; //First usage of "&": Use the address of an object
    object = 11.1;
    cout << "The object value of object_pointer pointing is: " << *object_pointer
    << endl;
    //Second usage of "*": Visiting the value of the object where pointer pointing at
    double& object_alia = object;
    //Second usage of "&": Creating a reference to an object
    cout << "The address of object is: " << &object << endl;
    cout << "The address of object_alia is: " << &object_alia << endl;
    cout << "The value of object_alia is: " << object_alia << endl;
    return 0;
}
```

```
The object value of object_pointer pointing is: 11.1
The address of object is: 012FF858
The address of object_alia is: 012FF858
The value of object_alia is: 11.1
```

Fig 12 Running result

13. * dereferencing operator, & address operator

“*” dereferencing is talked above in part 12. And “&” address operator is also discussed above in part 12.

14. how pointers work with built-in array, pointer arithmetic, pointer-based string

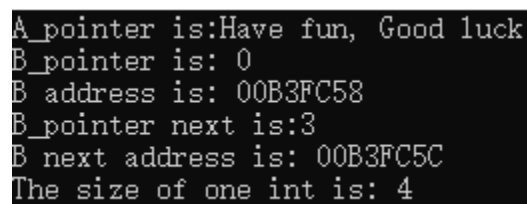
Example code:

```
int main(void)
{
    string A = "Have fun, Good luck";
    int B[6] = { 0, 3, 9, 8, 5, 2 };
    string* A_pointer;
    A_pointer = &A;
    int* B_pointer;
```

```

    B_pointer = B;
    cout << "A_pointer is:" << *A_pointer << endl;
    //A_pointer++; Compiler error happened
    //String pointer only have one pointer pointing at the whole string;
    cout << "B_pointer is: " << *B_pointer << endl;
    cout << "B address is: " << B_pointer << endl;
    B_pointer++; //Pointer arithmetic
    cout << "B_pointer next is:" << *B_pointer << endl;
    cout << "B next address is: " << B_pointer << endl;
    cout << "The size of one int is: " << sizeof(B[0]) << endl;
    return 0;
}

```



```

A_pointer is: Have fun, Good luck
B_pointer is: 0
B address is: 00B3FC58
B_pointer next is: 3
B next address is: 00B3FC5C
The size of one int is: 4

```

Fig 14 Running result

Let us just directly watch the running result of this code. I set a string as variable A and a pointer named A_pointer with it. The difference between string pointer and built-in array pointer is string pointer will just point to the whole string but the built-in array will just point at the beginning address of the array. If we try to visit the next address of the string pointer, compiler error will be thrown to us. If you still want to use string pointer to visit every single character in it, you can try to use char type array.

As for the built-in array pointer, I set it as B_pointer which will point at an int type array named B. B_pointer could do pointer arithmetic. The pointer will point at B[0] which is the starting address of the built-in array. And if we add 1 to the pointer, it will jump to the next variable in the array which is B[1]. Also, if we observe the address and the size of one block in int array, we will find the pointer will add one block size to the original address when it is added 1.

15. 4 cases in using const with pointers

Nonconst pointer to nonconst data: In this case both pointer and variable could be modified.

```
Int number;
```

```
Int* pointer=&number;
```

Nonconst pointer to const data: In this case, the pointer could be set to another variable but the variable with const cannot be modified.

```
Const int* pointer;
```

Const pointer to nonconst data: In this case, the data could be modified but the pointer will always point to the same address and cannot be modified.

```
Int* const pointer=&number;
```

Const pointer to const data: In this case, both the pointer and the data cannot be modified through the whole program.

Const int* const pointer=&number;

16. uses of static in variable declaration, in function declaration, in class variable/member function declaration

Use of static in variable declaration: When a variable is declared as static, it will not be destroyed until the end of the program. i.e. The variable will exist through the whole lifetime of a program

When it is declared in function, it will keep the rules that it will exist all the time. However, when the function call ends, this static variable will still exist. The value of it is kept but it cannot jump out of scope rule. We cannot visit it from outer scope.

When it is declared in class variable and class member function, please see the example code below:

```
class box
{
public:
    static int objectcount;//Creating static class member
    box()
    {
        objectcount++;
    }
    static int get_box_count()
    {
        return objectcount;
    }
};
int box::objectcount = 0;//Initial the box count value
int main(void)
{
    cout << "The number of boxes initialized are: " << box::get_box_count() << endl;

    //Usually we can visit the static class member via "::"
    box box_1;
    box box_2;
    cout << "The number of boxes are: " << box::get_box_count() << endl;
    //We can also use static member function via "::"
    //Static member function could only use static member variable, other
    //(Cont'd)static member function and other functions outside of the class
    return 0;
}
```

In this case I create a class named box which has static member objectcount and static member function get_box_count(). Static member variable is shared between every instance and there is only one copy. So, it is usually used to describe some common feature between different instances of the same class. Be careful that before you use your static member

variable you should initialize it first outside of the main function. The static member function is usually used to do some static member variable manipulation. Because as the annotation said, it could only visit the static member variable etc.

```
The number of boxes initialized are: 0
The number of boxes are: 2
```

Fig 16 Running result

17. uses of extern identifier to solve linker command failure (how global variable is declared, a demo in class 4)

Extern identifier is something more global than global identified. Global identified could only make the variable be accessed through the whole .cpp file. However, extern identifier could make the extern variable be accessed through the whole project files. For example: extern_example.h:

```
#ifndef EXTERN_EXAMPLE_H
#define EXTERN_EXAMPLE_H
extern int super;
#endif
```

extern_example.cpp:

```
#include "extern_example.h"
int super = 10;
```

main.cpp:

```
#include "extern_example.h"
#include<iostream>
int main(void)
{
    cout << "The value of super is: " << super << endl;
    return 0;
}
```

In this simple example you will find that we did not declare super in main function and we just use it. That's because we use extern identifier to it. This super variable could be accessed by all the programming files. But be careful that extern variable should be initialized in implementation file before any usage. When linker command failure happens, we can use extern identifier to find the extern variable outside of the .cpp file to solve the problem. Also, if we define an extern variable in main.cpp and we just use it in main.cpp there will be a linker error. That is because we said extern, which means the extern variable does not exist in the current file. In this case we can just delete extern or adding extern variable to other files to solve the problem.

```
The value of super is: 10
```

Fig 17 Running result

18. principle of least privilege

Every program and every user of the system should operate using the least set of privileges necessary to complete the job. Primarily this principle limits the damage that can result from an accident or error. The principle of least privilege (PoLP; also known as

the principle of least authority) is an important concept in computer security, promoting minimal user profile privileges on computers, based on users' job necessities.

In C++ programming, const identifier is a typical PoLP example. Const identifier will set some important variable kind of like read only. Also, some const function will not have the rights to change or modify any existed data.

In class private and public members are also examples for coder to understand how to use PoLP. Sometimes we just want to lock important variable in private which could only be accessed through member functions. So, in main function or other function these private members could not be directly accessed to keep the instances' data private.