

MODULE 20 STORAGE CLASSES, const, volatile, local and global

My Training Period: hours

Abilities

- Understand and use storage classes: auto, extern, static and register.
- Understand and use the const for variable and member function.
- Understand and use the volatile keyword.
- Understand the external and internal linkages terms.

20.1 Introduction

- Let figure out a typical C / C++ program that we have come across before, as shown below.
- This **program** is in one **file**. From this file we need other **files** such as headers and C / C++ resources files (such as libraries, object files etc). During the compile and run time, these entire resources linked together.
- There are also other resources needed dynamically during the program running such as input, dll files and for GUI programming, a lot more resources needed.

```
#include <iostream.h>
//Here, iostream.h, we have class declaration
//and implementation parts. We may have member
//variables, member functions, array, pointers,
//struct, enum, typedef, normal variables etc.
#define ...
//other variables...

class MyClass
{
};
//class variables...

union
{...};

struct struct1
{...};
//Another variable...

enum enum1 {...};
//Another variable...

inline int function1( )
{...}
//Another variables here...

void function1();
{...}
//Another variables here...

typedef R S;
//Another variables here...

//Other user defined data types...

int r, s, t;

main()
{
    struct X;
    enum Y;
    typedef P Q;
    union Z;
    class_type object1, object2;
    //class type variable or objects...
    ...

    int x, y;
    //other variables...
}
```

- You can see that it is very complex construction when we think about the variables. Hence, when declaring and defining the various variables with various data types in different locations, the **storage allocation, the lifetime, the visibility or accessibility** and how the **compiler treat the variables** is important to be concerned about.
- Keep in mind that, other than logical errors, most of the violations have been guarded by the compilers through warning and error messages. What a fortunate programmers we are!

20.2 Storage Classes

- Storage class specifiers tell compiler the duration and visibility of the variables or objects declared, as well as, where the variables or objects should be stored.
- In C++ program we have multiple files. In these files we may have normal variables, array, functions, structures, unions, classes etc. So, variables and objects declared must have the visibility, the lifetime and the storage, when values assigned.
- In C / C++ there are 4 different storage classes available: `automatic`, `external`, `static` and `register`. It is similar in C, explained somewhere in tenouk.com :o).

Storage class	Keyword
Automatic	<code>auto</code>
External	<code>extern</code>
Static	<code>static</code>
Register	<code>register</code>

Table 20.1: storage classes

- The general form for declaring a storage class is:

```
storage_class declarator;
```

- For example:

```
extern int value;
auto long p = 5;
auto int q;
static int x;
```

20.2.1 Automatic Variable - `auto`

- **Local** variables are variables declared within a function or blocks (after the opening brace, { of the block). Local variables are automatic by default. This means that they come to **existence** when the function in which it is declared is invoked and **disappears** when the function ends.
- Automatic variables are declared by using the keyword `auto`. But since the variables declared in functions are automatic by default, this keyword may be dropped in the declaration as you found in many source codes.
- The **same variable names** may be declared and used in different functions, but they are only known in the functions in which they are declared. This means, there is no confusion even if the same variables names are declared and used in different functions.
- Examples if we want explicitly declare the automatic type:

```
auto int x, y, z = 30;
auto char firstname;
```

- Same as:

```
int x, y, z = 30;
char firstname;
```

20.2.2 External Variable - `extern`

- External variables are variables that are recognized **globally**, rather than locally. In other words, once declared, the variable can be used in any line of codes throughout the rest of the program.

- A variable defined outside a function is external. An external variable can also be declared within the function that uses it by using the keyword `extern` hence it can be accessed by other code in other files.
- Program segment example:

```

int    value1;
char   name;
double value2;
//three externally defined variables

main()
{
    extern int value1;
    extern char name;
    extern double value2;
    //three externally defined variables
    //can be accessed from outside of this main()
    extern value3;
    //can be accessed from outside of this main()
    ...
}

```

- Note that the group of `extern` declarations may be omitted entirely if the original definition occurs in the **same** file and **before** the function that uses them.
- Therefore in the above example, the three `extern` declarations may be dropped. However, including the `extern` keyword explicitly will allow the function to use external variable even if it is defined later in a file or even in a different file provided both files will be compiled and linked together.

20.2.3 Static Variable - `static`

- In a single file program, static variables are defined within individual functions that they are local to the function in which they are defined. Static variables are local variables that **retain their values** throughout the lifetime of the program. In other words, their same (or the latest) values are still available when the function is re-invoked later.
- Their values can be utilized within the function in the same manner as other variables, but they cannot be accessed from outside of their defined function.
- The `static` has internal linkage (that is not visible from outside) except for the static members of a class that have external linkage. The example using the static variable has been presented in Module 13 and some other part of the program examples in this Tutorial.
- Simple program example:

```

//demonstrate the static variable...
#include <iostream.h>
#include <stdlib.h>

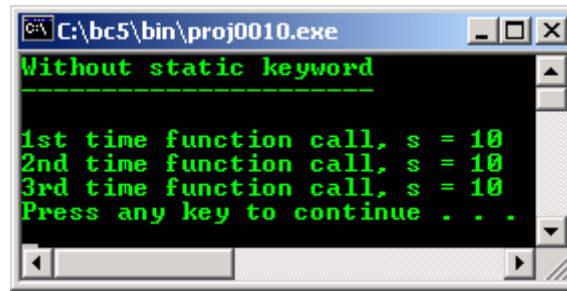
int funcstatic(int)
{
    //local variable should exist locally...
    int sum = 0;
    sum = sum + 10;
    return sum;
}

int main()
{
    int r = 5, s;

    //test the function calls several times...
    cout<<"Without static keyword\n";
    cout<<"-----\n\n";
    s = funcstatic(r);
    cout<<"1st time function call, s = "<<s<<endl;
    s = funcstatic(r);
    cout<<"2nd time function call, s = "<<s<<endl;
    s = funcstatic(r);
    cout<<"3rd time function call, s = "<<s<<endl;
    system("pause");
    return 0;
}

```

Output :



```
C:\bc5\bin\proj0010.exe
Without static keyword
-----
1st time function call, s = 10
2nd time function call, s = 10
3rd time function call, s = 10
Press any key to continue . . .
```

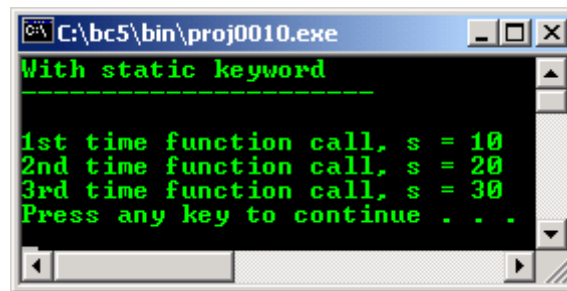
- Then change the following code in function `funcstatic()`:

```
int sum = 0;
```

- To:

```
static int sum = 0;
```

- Recompile and rerun the program, the output should be as shown below. By using the static variable, the previous value of the `sum` variable, by previous function call, is retained although it is just local variable.



```
C:\bc5\bin\proj0010.exe
With static keyword
-----
1st time function call, s = 10
2nd time function call, s = 20
3rd time function call, s = 30
Press any key to continue . . .
```

20.2.4 Register Variable - register

- The above three classes of variables are normally stored in computer memory. Register variables however are stored in the **processor registers**, where they can be accessed and manipulated faster. Register variables, like automatic variables, are local to the function in which they are declared.
- Defining certain variables to be register variables does not, however, guarantee that they will actually be treated as register variables.
- Registers will be assigned to these variables by compiler so long as they are available. If a register declaration cannot be fulfilled, the variables will be treated as automatic variables. So, it is not a mandatory for the compiler to fulfill the register variables.
- Usually, only register variables are assigned the register storage class. If all things equal, a program that makes use of register variables is likely to run faster than an identical program that uses just automatic variables.

20.2.5 Global Variables

- Global variables are defined outside of all function bodies (`{ . . . }`) and are available to all parts of the program. The lifetime or availability of a global variable last until the program ends.
- As explained before, if `extern` keyword is used when declaring the global variable, the data is available to this file by telling it the data is exist somewhere in another files.

20.2.6 Local Variables

- Local variables are local to a function, including the `main()` function. They are automatic variables, exist when the scope is entered and disappear when the scope closes.
- Let try some experiment. First of all, let create a simple class. Create a header file named `object.h`, save this file, do not run or compile this program.

- In this program, we declare global external variable `global1` as shown below:

```
extern int global1;

//Program object.h, the header file
//extern global variable
//external to object.cpp
extern int global1;

//---class declaration part---
class object
{
public:
    int objectvar;

public:
    object(void);
    int set(int);
    ~object(void);
};
//this header file cannot be compiled or run
```

- Next create the implementation file `object.cpp`, compile but do not run this program. In this program, we declare and define:

```
//extern...
int global1 = 30;
int global2 = 40;

//The class implementation file
//Program object.cpp
#include <iostream.h>
#include "object.h"

//extern...
int global1 = 30;
int global2 = 40;

//-----class implementation part-----
object::object(void)
{
    objectvar = 0;
}

int object::set(int newvalue)
{
    int local1 = 10;
    //non extern with same variables name....
    global1 = 60;
    global2 = 70;

    //Display the local variable locally...
    cout<<"In object.cpp file, local function, local1 variable = "<<local1<<endl;
    //Display the global variable locally...
    cout<<"In object.cpp file, global1 variable = "<<global1<<endl;
    cout<<"In object.cpp file, global2 variable = "<<global2<<endl;
    return newvalue;
}

object::~~object(void)
{
    objectvar = 0;
}
```

- And in local function of the `object.cpp`:

```
int object::set(int newvalue)
{...}
```

- We declare and define:

```
int local1 = 10;
//non extern, with same variables name....
global1 = 60;
global2 = 70;
```

- Finally create the main program, compile and run this program. In this program we declare and define:

```

    object    FirstObject;
    //external to object.cpp
    extern int global2;
    //local to this main() function...
    int local2 = 20;

//Program mainobject.cpp, here is the main program,
#include <iostream.h>
#include <stdlib.h>
#include "object.h"

main()
{
    object    FirstObject;
    //external to object.cpp
    extern int global2;
    //local to this main() function...
    int local2 = 20;

    cout<<"In object.h, global1 is object.cpp external variable = "<<global1<<endl;
    cout<<"In mainobject.cpp, global2 is object.cpp external variable
= " <<global2<<endl;
    cout<<"In mainobject.cpp, object value = " <<FirstObject.set(50)<<"\n";
    cout<<"In mainobject.cpp, local function, local2 variable = " <<local2<<endl;
    system("pause");
}

```

Output :

- Keep in mind that there is another layer of restriction in the class declaration using the public, protected and private, but for this example, all the restriction has been disabled by using the public keyword to keep it simple.

20.3 Constant Values - const

- In the most basic form, the const keyword specifies that a variable's value is constant and tells the compiler to prevent the programmer from modifying it.
- Example of the pointer declaration using const is shown below:

```

//Pointer to constant int
int const *PtrVar;

//Pointer to constant int
int const (*PtrVar);

//Constant pointer to int
int *const PtrVar;

//Constant pointer to int
int (*const PtrVar);

```

- Program example:

```

//const variable
#include <iostream>
#include <stdlib.h>

```

```

int main()
{
    //p = 10 is a constant value, cannot be modified
    //during the program execution...
    const int p = 10;

    cout<<"q = p + 20 = "<<(p + 20)<<" where, p = 10"<<endl;

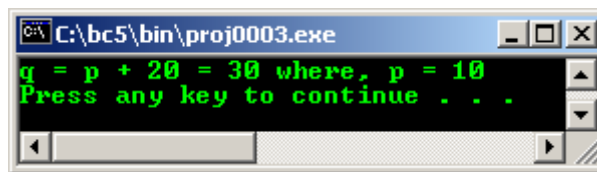
    //The following code should generate error, because
    //we try to modify the constant value...
    //uncomment, recompile notice the error...

    //p = 15;
    //--p;

    system("pause");
}

```

Output:



- We can use the `const` keyword instead of the `#define` preprocessor directive to define constant values.
- In C, constant values default to external linkage, so they can appear only in **source files** but in C++, constant values default to internal linkage, which allows them to appear in **header files**.
- The `const` also can be used in pointer declaration. A pointer to a variable declared as `const` can be assigned only to a pointer that is also declared as `const`.
- Another program segment example:

```

//const variable
#include <iostream>

const int ArrayOne = 64;
//The following code is legal in C++, but not in C
char StoreChar[ArrayOne];

int main()
{
}
//No output for this example

```

- Try another program example.

```

//a const pointer to a variable...
#include <iostream>
#include <stdlib.h>

int main()
{
    //declare the pointers and let they point
    //to something...
    //non const pointer...
    char *BuffOne = NULL, *BuffTwo = NULL;
    //a constant pointer...
    //assign the BuffOne pointer to PtrOne pointer
    char *const PtrOne = BuffOne;

    //Let it point to some data...
    *PtrOne = 'z';
    cout<<"The value pointed by constant pointer is "<<*PtrOne<<endl;

    //The following code will generate error, because we try to
    //assign non const pointer to const pointer...
    //PtrOne = BuffTwo;
}

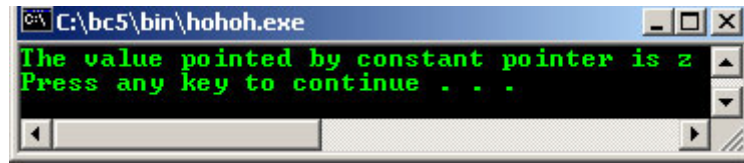
```

```

        system("pause");
        return 0;
    }

```

Output:



- Uncomment the `//PtrOne = BuffTwo` code and recompile the program, something like the following error should be generated.

```
Error: project.cpp(17, 10):Cannot modify a const object
```

```

//a pointer to a const variable...
#include <iostream>
#include <stdlib.h>

int main()
{
    const char *BuffOne = "Testing";

    cout<<"The data pointed by BuffTwo is "<<BuffOne<<endl;

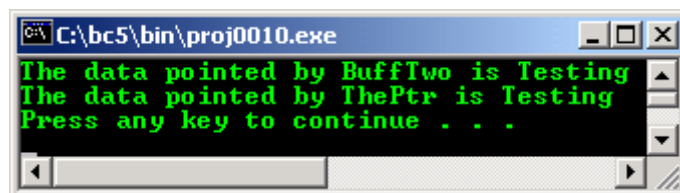
    //The const pointer BuffOne assigned to the
    //const pointer ThePtr is OK...
    const char *ThePtr = BuffOne;
    cout<<"The data pointed by ThePtr is "<<ThePtr<<endl;

    //The following code will generate an error
    //cannot modify the const....
    /*ThePtr = 'z';

    system("pause");
    return 0;
}

```

Output:



- Uncomment the code `/*ThePtr = 'z';` and recompile the program, the same error as in the previous example should be expected.
- The const declaration also normally used in the definition of a function's arguments, to indicate it would not change them as shown below making the code clearer and to avoid error.

```
int strlen(const char []);
```

20.3.1 Constant Member Function

- When declaring a member function with the `const` keyword, this specifies that it is a **read only** function that does not modify the **object** (notice the differences between variable versus **object**) for which it is called.
- A constant member function cannot modify any data members or call any member functions that are not constant.
- Implicitly, the `const` has set the 'can't modify' `*this` pointer. This can be changed by using the `mutable` (preferred) or `const_cast` operator.
- Pointer to constant data can be used as function parameters to prevent the function from modifying a parameter passed through a pointer.

- Place the `const` keyword after the closing parenthesis of the argument list.
- `const` keyword is required in both the **declaration** and the **definition**.
- Program example:

```
//constant member function
#include <iostream>
#include <stdlib.h>

//-----Class declaration part-----

class Date
{
    int month;

    public:
    //we would test the month only...
    Date (int mnt, int dy, int yr);
    //A write function, so can't be const
    void SetMonth(int mnt);
    //A read only function declaration
    int GetMonth() const;
};

//-----Class implementation part-----

Date::Date(int,int,int)
{
}

void Date::SetMonth(int mnt)
{
    //Modify the non const member variable data
    month = mnt;
}

//A read only function implementation
int Date::GetMonth() const
{
    //Does not modify anything
    return month;
}

//-----main program-----
void main()
{
    Date TheDate(7,4,2004);
    //non const member function, OK
    TheDate.SetMonth(11);

    cout<<"Month of the sample date is "<<TheDate.GetMonth()<<endl;

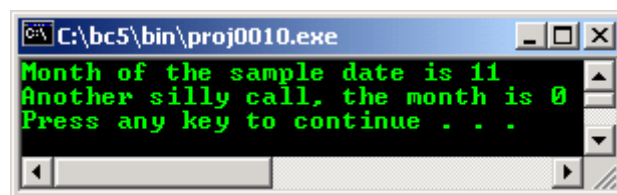
    //another dummy const object...
    const Date BirthDate(7,4,1971);
    //Then try to modify the const object, NOT OK
    //BirthDate.SetMonth(5);

    //const member function sending message...
    BirthDate.GetMonth();
    //So, the following shouldn't have the output data...
    cout<<"Another silly call, the month is "<<BirthDate.GetMonth()<<endl;

    system("pause");
}

```

Output:



```
C:\bc5\bin\proj0010.exe
Month of the sample date is 11
Another silly call, the month is 0
Press any key to continue . . .
```

- Uncomment the `//BirthDate.SetMonth(5);`, then rerun the program, an error something like the following statement should be expected.

Non-const function Date::SetMonth(int) called for const object

- The const-ness of the function can be disabled by using the mutable keyword. The program example will be presented in another Module.

20.4 volatile

- It is a type qualifier used to declare an object or variable value that can be modified by **other** than the statement in the source codes itself, such as interrupt service routine and memory-mapped I/O port or concurrent thread execution.
- Keep in mind that although we have to concern about these volatile variable or object, most of the compilers nowadays have their own implementation how to handle this situation mainly for Win32 applications.
- For example if you want to create multithreaded program, there are C++ compiler or project settings for multithreaded program. You have to check your compiler documentation.
- When declaring an object to be volatile, we tell the compiler not to make any assumptions concerning the value of the object while evaluating expressions in which it occurs because the value could change at any moment.
- When a name is declared as volatile, the compiler reloads the value from memory each time it is accessed by the program. Volatile codes will not be optimized by compiler to make sure that the value read at any moment is accurate.
- Without optimization, for example permitting the redundant reads, the volatile may have no effect.
- The keyword volatile is used **before** or **after** the data type declaration. They cannot appear after the first comma in a multiple variable declaration. For example:

- Declaring volatile variable. For example, volatile integer.

```
volatile int Vint;  
int volatile Vint;
```

- Declaring pointers to volatile variables. For example, pointer to volatile integer.

```
volatile int * Vintptr;  
int volatile * Vintptr;
```

- Declaring volatile pointer to non volatile variables. For example, volatile pointer to integer.

```
int * volatile Vptr;
```

- Declaring volatile pointer to a volatile variable. For example volatile pointer to volatile integer.

```
int volatile * volatile Vptr;
```

- Using typedef example.

```
typedef volatile int Vint
```

- Illegal declaring after the first comma in multiple variable declaration example.

```
int p, volatile Vint;
```

- When volatile is applied to the struct or union, the entire contents of the struct or union become volatile however we can also apply the volatile to the members of struct or union individually.
- volatile also applied to classes and their member functions.

20.5 Linkage

- Linkage is the process that allows each identifier to be associated correctly with one particular object or function for the sake of the linker during the linking process. Each identifier is allocated a memory storage that holds the variables or objects.
- This process must be obeyed to avoid the problem arises when the same identifier is declared in different scopes such as, in different files, or declared more than once in the same scope.

- All identifiers have one of three linkage attributes, closely related to their scope:
 - External linkage,
 - Internal linkage, or
 - No linkage.
- These attributes are determined by the **location** and **format** of the declarations, together with the explicit (or implicit by default) use of the **storage class specifier** `static` or `extern`.
- Each instance of a particular identifier with **external linkage** represents the same object or function throughout the entire set of files and libraries making up the program.
 - If the declaration of an object or function identifier contains the storage class specifier `extern`, the identifier has the same linkage as any visible declaration of the identifier with file scope. If there is no such visible declaration, the identifier has external linkage.
 - If an object identifier with file scope is declared without a storage class specifier, the identifier has external linkage.
- Each instance of a particular identifier with **internal linkage** represents the same object or function within one file only.
 - Any object or file identifier having file scope will have internal linkage if its declaration contains the storage class specifier `static`.
 - For C++, if the same identifier appears with both internal and external linkage within the same file, the identifier will have external linkage. In C, it will have internal linkage.
 - If a function is declared without a storage class specifier, its linkage is determined as if the storage class specifier `extern` had been used.
- Identifiers with **no linkage** represent unique entities.
 - Any identifier declared to be other than an object or a function (for example, a `typedef` identifier).
 - Function parameters.
 - Block scope identifiers for objects declared without the storage class specifier `extern`.
- Program example compiled using [VC++ / VC++ .Net](#).

```

//demonstrate the static variable...
#include <iostream>
using namespace std;

int funcstatic(int)
{
    //local variable should exist locally...
    static int sum = 0;
    sum = sum + 10;
    return sum;
}

int main()
{
    int r = 5, s;

    //test the function calls several times...
    cout<<"Without static keyword\n";
    cout<<"-----\n\n";
    s = funcstatic(r);
    cout<<"1st time function call, s = "<<s<<endl;
    s = funcstatic(r);
    cout<<"2nd time function call, s = "<<s<<endl;
    s = funcstatic(r);
    cout<<"3rd time function call, s = "<<s<<endl;
    return 0;
}

```

Output:

```
C:\ "g:\vcnetprojek\searchpat...
Without static keyword
1st time function call, s = 10
2nd time function call, s = 20
3rd time function call, s = 30
```

-----o0o-----

Further reading and digging:

1. [Check the best selling C / C++, Object Oriented and pattern analysis books at Amazon.com.](#)