**Storage Classes**

Storage Classes are used to describe the scope, visibility and life-time of a variable/function.

Specifiers precede the type that they modify

Storage classes

1. auto
2. register
3. static
4. extern
5. mutable // not available in C

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Storage  Class | Storage | Initial value | Scope | Lifetime |
| Automatic | Stack | Garbage | Within block | Function block |
| External | Data Segment | Zero | Global MF | Whole Program |
| Static | Data Segment | Zero | Within block | Whole Program |
| Register | CPU register | Garbage | Within block | Function block |
| Mutable | Mutable | - | Class | Class |

MF - Multiple File

# Automatic (auto)

* default storage class for all the variables
* auto can be used inside a function or a block, i.e. local varibales
* visibility - restricted to the function in which it is declared
* lifetime - limited till the time its container function is executing
* It can be accessed within nested blocks within the parent block/function in which the auto variable was declared
* It can be accessed outside their scope as well using the concept of pointers given here by pointing to the very exact memory location where the variables resides
* It’s default value is garbage
* It is created as soon as its declaration statement is encountered and is destroyed as soon the program control leaves its container function block

int x; // by default storage class is auto

auto int y;

auto int m = 10;

NOTE:

* Since C++11 auto is used as a declaration specifier
* Storage in primary memory

# Register

* To define local variables that should be stored in a register instead of RAM
* Max size is equal to the register size (usually one word)
* Unary operator ‘&’ can be applied to it (as it does not have a memory location)
* Objective is to increase its access speed which eventually makes the program run faster
* It might be stored in a register depending on hardware and implementation restrictions
* If there are no registers vacant to accommodate the variable then it is stored just like any other automatic variable
* Register keyword can be used with pointer variables

register int \* ptr = &i;

* Only register is allowed in a fun parameter declaration

**Example**

#include <stdio.h>

int main() {

auto int m = 10;

register int x = 5;

printf("m : %d\n", m);

printf("&m : %d\n\n", &m);

printf("x : %d\n", x);

//printf("&x : %d\n\n", &x);

// error: address of register variable 'x' requested

return 0;

}

Output:

m : 10

&amp;m : 372152084

x : 5

# Static

Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve their previous value in their previous scope and are not initialized again in the new scope.

* A static variable has visibility of a local variable and the lifetime of an external variable
* A static variable remains in memory while the program is running.
* Static variables are allocated memory in data segment, not stack segment.
* Static variables (like global variables) are initialized as 0 if not initialized explicitly.
* In C, static variables can only be initialized using constant literals
* In C, a value which is not known at translation time cannot be used for initialization of static variables
* In C++, static variables can be initialized at run time
* Static global variables and functions are also possible in C/C++.
* Static variables should not be declared inside structure. The reason is C compiler requires the entire structure elements to be placed together (i.e.) memory allocation for structure members should be contiguous.

**Note:** It is possible to declare structure inside the function (stack segment) or allocate memory dynamically (heap segment) or it can be even global (BSS or data segment). Whatever might be the case, all structure members should reside in the same memory segment because the value for the structure element is fetched by counting the offset of the element from the beginning address of the structure.

**Usage:**

* Mostly used to save values in a recursive function
* To limit scope of a variable or function within a file only
* Static storage class cannot be used with function parameter

Example

|  |  |  |
| --- | --- | --- |
| #include <stdio.h>  static int var;  static int var;  int main(void) {  int var = 7;  printf("%d", var);  return 0;  }  Output:  7 | #include <stdio.h>  static int var = 7;  static int var;  int main(void) {  printf("%d", var);  return 0;  }  Output:  7 | #include <stdio.h>  static int var = 3;  static int var;  int main(void) {  var = 7;  printf("%d", var);  return 0;  }  Output:  7 |

# External (extern)

* The extern keyword extends the visibility of the C variables and C functions.
* Also called global variable
* Declared outside a function but is accessible inside the function block
* Visibility - all across the program
  + - Accessible by all the functions present in the program
* Lifetime - lifetime of an external variable is same as the lifetime of a program
* A declaration can be done any number of times but definition only once.
* The extern keyword is used to extend the visibility of variables/functions.
* Since functions are visible throughout the program by default, the use of extern is not needed in function declarations or definitions. Its use is implicit.

int foo(int arg1, char arg2); // by programmer

extern int foo(int arg1, char arg2); // by compiler

* When extern is used with a variable, it’s only declared, not defined.
* As an exception, when an extern variable is declared with initialization, it is taken as the definition of the variable as well.

|  |  |  |
| --- | --- | --- |
| #include <stdio.h>  extern int var;  extern int var;  int main(void) {  int var = 7;  printf("%d", var);  return 0;  }  Output:  7 | #include <stdio.h>  extern int var = 7;  // warning: 'var' initialized and declared 'extern'  int main(void) {  printf("%d", var);  return 0;  }  Output:  7 | #include <stdio.h>  extern int var;  int main(void) {  var = 7; // CE: undefined reference to `var'  printf("%d", var);  return 0;  }  Output: |

# Mutable // not available in C

Applies only to class objects

Allows a member of an object to override constness

A mutable member can be modified by a const member function

**Example**

#include <iostream>

using namespace std;

class test {

mutable int a;

int b;

public:

test(int x, int y) {

a = x; b = y;

}

void square\_a() const {

a = a \* a;

}

void display() const {

std::cout << "a : " << a << std::endl;

std::cout << "b : " << b << std::endl;

}

};

int main(void) {

const test x(2, 3);

std::cout << "Initial value: " << std::endl;

x.display();

x.square\_a();

std::cout << "Final value: " << std::endl;

x.display();

return 0;

}

Output:

Initial value:

a : 2

b : 3

Final value:

a : 4

b : 3

# Automatic Type Specifier

* Since C++11 auto keyword is no longer a storage class specifier
* It acts as a type specifier that directs the compiler to deduce the type of a declared variable from its initialization expression
* Compiler deduces the type of an auto variable from the type of its initializer expression

auto i = 1.1; // i: double

* For variables, specifies that the type of the variable will be automatically deduced from its initializer
* For functions, specifies that the return type is a trailing return type or will be deduced from its return statements (C++11)
* For non-type template parameters, specifies that the type will be deduced from the argument (C++17)
* In a function declaration that uses the trailing return type syntax auto does not perform automatic type detection. It only serves as a part of the syntax.
* In a function declaration that does not use the trailing return type syntax auto indicate that the return type will be deduced from the operand of its return statement using the rules for template argument deduction.
* In case of multiple variable declaration, the type of each declarator can be deduced independently
* If deduced type is not the same in each deduction the program is ill-formed

auto I = 5, \*p = &I; // well –formed (int)

~~auto x = 3, y = 2.2; // ill – formed (error)~~

* Auto can be applied to new expression

auto \* p = new auto(1);

* Auto cannot deduce array types

char a[5];

~~auto b[5] = a;~~

* Auto cannot be used in function parameters

~~int func(auto x = 3);~~

**Example**

#include <iostream>

using namespace std;

int func(auto x = 3) { // 1

std::cout << "x: " << x << std::endl;

}

int main() {

auto i = 5, \*p = &i;

auto x = 3, y = 2.2; // 2

char a[5];

auto b[5] = a; // 3

return 0;

}

// 1 error: use of 'auto' in parameter declaration only available with -std=c++14 or -std=gnu++14

// 2 error: inconsistent deduction for 'auto': 'int' and then 'double'

// 3 error: 'b' declared as array of 'auto'

# END