**Advanced Features in C++**

Credit: Many examples are extracted from Geek for Geek

**(1) enum for enumeration**

Enumerated type

(enumeration) is a user-defined data type which can be assigned some limited values. These values are defined by the programmer at the time of declaring the enumerated type. Normally, the values are character strings without quotes. We use these strings as a better way to represent choices.

**Enum** keyword is used to declare enumerated types after that enumerated type name was written then under curly brackets possible values are defined. After defining Enumerated type variables are created.

Example 1: Case without enum. Here, choices 0, 1, and 2 are difficult to remember.

int class;

if(class ==0)

cout << "cis 200" << endl;

else if(class == 1)

cout << "cis 275" << endl;

else if(class == 2)

cout << "cis 210" << endl;

Example 2: Case with enum. Here, choices CIS210, CIS200, and CIS275 are easy to understand

enum Class2 {CIS210, CIS200, CIS275};

Class2 x;

x = CIS200;

switch(x)

{

case CIS210: // case 0:

...

case CIS200: // case 1:

...

case CIS275: // case 2:

...

}

**(2) union**

A union is a special data type available in C/C++ that allows storing different data types in the same memory location. The main purpose of introducing union is to save memory in the old days. The restriction of using a union is that at any moment only one data member of the union should be used. In other words, no two or more data members should be used at any time.

Example 1: struct

struct X

{

int a;

double b;

};

X v1;

v1.a = 10; v1.b=20.1;

Example 2: union

union Y

{

int a;

double b;

};

Y u={18, 38, "geeksforgeeks"};

printf("\nunion data:\n integer: %d\n "

"decimal: %.2f\n name: %s\n",

u.integer, u.decimal, u.name);

Output:

union data:

integer: 18

decimal: 0.00

name: ?

strcpy(u.name, "geeksforgeeks");

printf("\nunion data:\n integeer: %d\n "

"decimal: %.2f\n name: %s\n",

u.integer, u.decimal, u.name);

output:

union data:

integeer: 1801807207

decimal: 277322871721159510000000000.00

name: geeksforgeeks

**(3) auto keyword**

The auto data type is inferenced from the context.

When initializing a variable, the auto keyword can be used in place of the type to tell the compiler to infer the variable’s type from the initializer’s type. This is called type inference (also sometimes called type deduction).

Example 1:

auto d{ 5.0 }; // 5.0 is a double literal, so d will be type double

auto i{ 1 + 2 }; // 1 + 2 evaluates to an int, so i will be type int

Example 2:

auto add(int x, int y) // auto 🡪 int

{

return x + y;

}

**(4) volatile keyword**

This keyword is used to prevent the compiler from optimizing the code related to the volatile variable.

Objects declared as volatile are omitted from optimization because their values can be changed by code outside the scope of current code at any time:

1. Global variables modified by an interrupt service routine outside the scope:

For example, a global variable can represent a data port (usually global pointer referred as memory mapped IO) which will be updated dynamically.

1. Global variables within a multi-threaded application

Example 1:

volatile int x = 10;

**(5) register keyword**

Registers are faster than memory to access, so the variables which are most frequently used in a C program can be put in registers using register keyword.

It tells the compiler to put the related variable into a register instead of a regular memory location.

The main purpose is to reduce the memory access time for this particular variable. It is commonly used for real-time applications.

Example 1:

register int j = 10;

**(6) static keyword**

Case 1: Inside a regular function, a static variable is history-sensitive. In other words, C++ remembers the value of this static variable from its previous function call.

int getMax(int x)

{

static int max = 0; // max becomes history-sensitive

if(x> max)

max = x;

return max;

}

Case 2: Inside a class, a static data member or member function has only one copy shared by all the objects of this class.

class Student

{

public:

static int x; // there is only one copy of x for all the

// objects of class Student

int y; // Each object of class Student has a different copy

// variable y

static void foo( ) // only one copy for all the instances of

// class Student

{

// only local and static variables can be used here

}

....

};

int main()

{

Student a, b;

a.x =10; b.x=20;

a.y =10; b.y=20;

cout << a.x << endl; //20

cout << a.y << endl; // 10

…

}

**(7) extern keyword**

Case 1: extern variables

File1.cpp:

Int x;

……

File2.cpp:

Extern int x;

…..

Case 2: C functions

// Save file as .cpp and use C++ compiler to compile it

extern "C"

{

int printf(const char \*format,...);

}

int main()

{

printf("GeeksforGeeks");

return 0;

}

#ifdef \_\_cplusplus

extern "C" {

#endif

/\* Declarations of this file \*/

#ifdef \_\_cplusplus

}

#endif

**(8) Preprocessor**

Preprocessors are programs that process our source code before compilation.

Case 1: macro

// macro with parameter

#define AREA(l, b) (l \* b)

Case 2: Conditional compilation

Conditional Compilation directives are type of directives which helps to compile a specific portion of the program or to skip compilation of some specific part of the program based on some conditions.

#ifdef macro\_name

statement1;

statement2;

statement3;

.

.

.

statementN;

#endif

#undef macro\_name

Case 3: Once-only Header File

/\* File foo.h \*/

#ifndef FILE\_FOO\_SEEN

#define FILE\_FOO\_SEEN

the entire file

#endif /\* !FILE\_FOO\_SEEN \*/

File1.cpp:

#include “foo.h”

…

File2.cpp:

#include “foo.h”

…

**(9) Nested Class**

A nested class is a class which is declared in another enclosing class. A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed.

Example 1:

#include<iostream>

using namespace std;

/\* start of Enclosing class declaration \*/

class Enclosing {

private:

int x;

/\* start of Nested class declaration \*/

class Nested {

int y;

void NestedFun(Enclosing \*e) {

cout<<e->x; // works fine: nested class can access

// private members of Enclosing class

}

}; // declaration Nested class ends here

}; // declaration Enclosing class ends here

int main()

{

}

Example 2:

#include<iostream>

using namespace std;

/\* start of Enclosing class declaration \*/

class Enclosing {

int x;

/\* start of Nested class declaration \*/

class Nested {

int y;

}; // declaration Nested class ends here

void EnclosingFun(Nested \*n) {

cout<<n->y; // Compiler Error: y is private in Nested

}

}; // declaration Enclosing class ends here

int main()

{

}