Struct in C++

For C++, the only difference between *struct* and *class* is that by default in *struct* the members are *public* and in class *private*.

```
class Date
{
    int Day, // no access modifier, consequently private
    iMonth,
    Year;
    Date() { } // error – private constructor is useless
    ……….
};
struct Date
{
    int Day, // no access modifier, consequently public
    iMonth,
    Year;
    Date() { }
    ……….
};
```

In practice, however, the *struct* is used for small classes containing only attributes. Or in other words – *struct* in C++ has the same meaning as in C.

Copy constructor (1)

```
class Date
{
   int Day, iMonth, Year;
   public: Date(int, int, int);
    ……..……..
};
void PrintDate(Date d)
{
  printf("%d-%d-%d\n", d.GetDay(), d.GetMonth(), d.GetYear());
}
```

Each class has default copy constructor that copies byte by byte from original object into the new object:

```
Date d1(27, 5, 2019);
Date d2 = d1; // d2 is created by default copy constructor
Date *pd3 = new Date(27, 5, 2019);
Date d4 = *pd3; // d4 is created by copy constructor, pd3 points to the original
PrintDate(d1);  // argument d is created by copy constructor from d1
PrintDate(*pd3); // argument d is created by copy constructor, pd3 points to the original
PrintDate(Date(27, 5, 2019)); // contructs an object without name
                                    // this nameless object is the original in creating d
```

Copy constructor (2)

```
class Date
{
   int Day;
   char *pMonth = 0;
   int Year;
public:
    Date(int, const char *, int);
    ~Date() {if (pMonth) delete pMonth; }
……….
};
```

Let:

```
static Date d1(27, "May", 2019); // global lifetime
Date d2 = d1; // local lifetime
```

Problem: as the default copy constructor copies attribute by attribute, the pointers *d1.pMonth* and *d2.pMonth* point to the same memory field. Consequently, when *d2* as a local variable is deleted, *d1* looses its value for attribute *pMonth*.

To overcome this and similar problems, we have to write our own copy constructor.

Copy constructor (3)

```
class Date
{
  int Day;
  char *pMonth = 0;
  int Year;
public:
        Date(int, const char *, int);
        ~Date() { delete pMonth; }
        Date (const Date &Original)
        {  // overloads the default copy constructor
                Day = Original.Day; Year = Original.Year;
                int n;
                pMonth = new char[n = strlen(Original.pMonth) + 1];
                strcpy_s(pMonth, n, Original. pMonth);
        }
        ……..
};
Date d2 = d1; // When the copy constructor is working, Original is the synonym of d1;
                  // Day, Year and pMonth are the members of d2.
```

Pointer *this*

By default, each class has a member called as *this*. It is the pointer which points to the current object itself. For example:

```
class Date
{
    ……..
    Date (const Date &Original)
    {  // overloads the default copy constructor
        *this = Original; // At first copy everything, then modify
                          // Default assignment operator (discussed later) is applied
       int n;
        pMonth = new char[n = strlen(Original.pMonth) + 1];
        strcpy_s(pMonth, n, Original.pMonth);
    }
    int GetDay() { return this->Day; }
     // some programmers mark all the methods and variables that are class members
     // with this->. Reason: they want to distinguish the local variables from class
     // members
    ……..
};
```

Friends (1)

Let us have class Date and the similar to it class Time:

```
class Time {
  int Hour, Min, Sec;
public:
  Time(int, int, int);
  ……….……
};
```

Let us have also class Timestamp:

```
class Timestamp {
Date date; // aggregation
Time time;
public:
Timestamp();
~Timestamp();
Timestamp(int d, int mn, int y, int h, int m, int s) : date(d, mn, y), time(h, m, s) { }
void PrintTimestamp() {
      printf("%02d-%d-%d %02d:%02d:%02d\n", date.GetDay(), date.GetMonth(),
            date.GetYear(), time.GetHour(), time.GetMinutes(), time.GetSeconds());
      }
};
```

Friends (2)

If the author of all these 3 classes is the same programmer, then in class *Timestamp* he would like to access the members of *Date* and *Time* directly. In C++ it is possible if classes *Time* and *Date* are declared as friend classes of *Timestamp* :

```
class Time {
   ……….
   friend class Timestamp;
};
class Date {
   ……….
   friend class Timestamp;
};
```

Now:

```
class Timestamp {
  ……….
void PrintTimestamp() { // accessor functions not needed
      printf("%02d-%d-%d %02d:%02d:%02d\n", date. Day, date.iMonth,
               date. Year, time. Hour, time.Min, time. Sec);
      }
};
```

Friends (3)

If class A declares that class B is its friend, class B has free access to all the members of class A. But it does not mean that A can also access non-public members of B. Here classes *Time* and *Date* allow class *Timestamp* to work with its private attributes. As *Timestamp* has not declared friendship with Time and Date, those classes have no free access to *Timestamp* private and protected members.

Friendship is not inherited. Also, if B declares that C is its friend, C has access to non-public members of B but not to non-public members of A.

A class may also declare that a function out of classes is its friend. Example:

```
class Time; // put it at the beginning of file Date.h to explain the compiler
            // the meaning of word Time
class Date {
……….
friend void PrintTimestamp(Date *, Time *);
};
class Date; // put it at the beginning of file Time.h to explain the compiler
            // the meaning of word Date
class Time {
……….
friend void PrintTimestamp(Date *, Time *);
};
```

Friends (4)

Now function *PrintTimestamp* has access to all the members of classes *Date* and *Time*:

```
void PrintTimestamp(Date *pd, Time *pt)
{
   printf("%02d-%d-%d %02d:%02d:%02d\n", pd->Day, pd->iMonth, pd->Year,
                                          pt->Hour, pt->Min, pt->Sec);
}
```

Usage:

```
Date d(8, 3, 2019);
Time t(11, 3, 56);
PrintTimestamp(&d, &t);
```

Operator overloading (1)

```
class complex
{
  public:  double Re, Im; // real part and imaginary part
           complex(double d1 = 0, double d2 = 0) { Re = d1;  Im = d2; }
           complex operator+(complex &c)
                    { return complex (Re + c.Re, Im + c.Im); }
           int operator==(complex &c)
                    { return Re == c.Re && Im == c.Im ? 1 : 0; }
           complex operator!() { return complex(Re, -Im); }
           ……..
};

complex x(5, 6), y(1,2); // x = 5 + j6, y = 1 + j2
complex z1 = x + y; // Actually z1 = x.operator+(y);  we get z1 = 6 + j8.
           // When the operator method is working, c is the synonym of y; Re and
           // Im are the members of x.  The return value is a new nameless
           // complex  number. From it the default copy constructor creates z1.
if (x == y) // actually x.operator==(y)
   printf("Equal\n");
complex z2 = !x;  // actually z2 = x.operator!(); we get z2 = 5 - j6 (conjugate of x)
```

Operator overloading (2)

Alternative solution:

```
class complex {
  public:  double Re, Im;
           complex(double d1 = 0, double d2 = 0) { Re = d1;  Im = d2; }
           friend complex operator+(complex &, complex &);
           friend int operator==(complex &, complex &);
           friend complex operator!(complex &);
           ……..
};
complex operator+(complex &a, complex &b) {
    return complex(a.Re + b.Re, a.Im + b.Im);
}
int operator==(complex &a, complex &b) {
     return (a.Re == b.Re && a.Im == b.Im) ? 1 : 0;
}
complex operator!(complex &a) {
  return complex(a.Re, -a.Im);
}
```

Operator overloading (3)

```
complex x(5, 6), y(1,2); // x = 5 + j6, y = 1 + j2
complex z1 = x + y; // actually z1 = operator+(x, y);  we get z1 = 6 + j8
                    // complex operator+(complex &a, complex &b) {
                    // return complex(a.Re + b.Re, a.Im + b.Im); }
                    //
                    // When the operator method is working, a is the synonym of
                    // x and b is the synonym of y. The return value is a new
                    // nameless complex  number. From it the default copy
                    // constructor creates z1.
if (x == y) // actually operator==(x, y)
     printf("Equal\n");
complex z2 = !x;  // actually z2 = operator!(x); we get z2 = 5 - j6
```

Operator overloading (4)

It is not possible to:

1. Introduce new operators not specified in C++ standard.
2. Change the priorities.
3. Overload the *sizeof* operator, the scope resolution operator (*::*), the conditional operator (*?:*) and the member selection operator (.).

Overloading of operators like *new*, *delete*, function call (*( )*), array element reference (*[ ]*), comma (,), assignment (=) and type cast may be tricky.

Let us take class *Date*:

Date d1(20, 10, 2019); // constructor called
Date d2 = d1; // default copy constructor called
Date d3; // constructor without arguments called
d3 = d1; // here we need operator overloading function for assignment

Each class has default assignment overloading function providing byte-by-byte copy. Rather often it is not acceptable and we have to write our own assignment overloading function replacing the default one.

Operator overloading (5)

Let us have:

```
class Date {
   int Day;
   char *pMonth = 0;
   int Year;
public:
    Date() { }
    Date(int, const char *, int);
    ~Date() { if (pMonth) delete pMonth; }
……….
};
```

Let:

```
Date *pd1 = new Date(8, "March", 2019);
Date *pd2 = new Date; // constructor without arguments called
*pd2 = *pd1; // default assignment overloading function called
…….
delete pd1;
```

Problem: as the default assignment overloading function copies attribute by attribute, two objects of class *Date* share common memory field for month.. Consequently, deleting one of them corrupts the other.

Operator overloading (6)

```
Date &Date::operator =(const Date &Right)  // here & - specifies the reference type
{
  if (this == &Right) // here & - address operator
      return *this; // necessary for expressions like d1 = *pd where pd points to d1
   Day = Right.Day;  Year = Right.Year;
   if (pMonth)
      delete pMonth;
   int n;
   pMonth = new char[n = strlen(Right.pMonth) + 1];
   strcpy_s(pMonth, n, Right. pMonth);
   return *this;
}
```

d1 = d2; // actually d1.operator=(d2);

i.e. *this* points to *d1* and *Right* is the synonym of *d2*

d1 = d2 = d3; // d1 = d2.operator=(d3) → d1.operator=(d2.operator=(d3));

Therefore void Date::operator=(Date &Right) {…} does not work – the operator= function must return the object.

Operator overloading (7)

```
class Date {
private:    int Day, Year;
            char *pMonth = 0, *pText = 0;
public:     ……….
            operator char *()     // operator function to overload type casting
                                  // no return value, the word "operator" is followed
                                  // by the new type specifier
            {
               pText = new char[64];
               sprintf_s(pText, 64, "%d %s %d", Day, pMonth, Year);
               return pText;
            }
};
Date d (27, "May", 2020);
if (strcmp(d, "28 June 2020")) {
        // actually the operator char *() function associated with object d is called
   printf(“Do not match\n”);
}
printf("%s\n", (char *)d);
```

Static members(1)

```
class Base
{
public:     static int Counter;   // declaration, but definition for initialization is also needed
            Base() { Counter++; }
            ~Base() { Counter--; }
};
int Base::Counter = 10; // definition, must be outside of functions and class declarations
                                    // applicable to public, protected and private members
Base b;
printf("%d\n", b.Counter);  // not recommended
printf("%d\n", Base:: Counter); // correct
```

Static attributes get memory only once. They are shared between all the objects of that class and also objects of classes derived from that class. The static attributes exist even when there are no any objects defined yet.

```
class Derived : public Base {……..};
Derived d;
printf("%d\n", d.Counter); // not recommended
printf("%d\n", Derived::Counter); // correct
```

Here *Counter* presents the current total number of objects of class *Base* plus objects of class *Derived*.

Static members(2)

```
class Base
{
private:    static int Counter;
public:      Base() {Counter++; }
             ~Base() {Counter--; }
             static int GetCounter() { return Counter; }
};
int Base::Counter=0; // although private
class Derived : public Base {……..};
Derived d;
printf("%d\n", d.GetCounter()); // not recommended
printf("%d\n", Derived::GetCounter()); // correct
printf("%d\n", Base::GetCounter()); // correct
```

Static functions of a class cannot operate with non-static members of that class. They can be called even when there are no any objects defined yet.

All the non-static functions have access to any of the static members, the restrictions depend only on the access specifiers (*public*, *private*, *protected*).

Constant members

The value of static or non-static value attribute may be declared as constant. In that case they must be initialized right in the declaration. Later changes, of course, are not possible.

Example:

```
class Date
{
 ……….
 const char MonthNames[12][4] = {
                                    "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                                    "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
                                  };
 ……….
};
```

Constant objects (1)

Let us have

```
class Test
{
private:
    int Value;
    int Counter;
public:
   Test(int i) { Value = i; Counter = 0; }
   void SetValue(int i) { Value = i; }
   int GetValue() { return Value; }
};
```

and

```
const Test *pt = new Test(1);
```

In that case:

```
pt->SetValue(5); // error: the object and consequently its attributes are constants
int i = pt->GetValue(); // also error!
```

If an object is constant, it is not possible to call methods associated with it.

Constant objects (2)

Solution:

```
int GetValue() const { return Value; } // const member function
```

Now:

```
const Test *pt = new Test(1);
int i = pt->GetValue(); // works
```

But the *const* member function cannot so simply change the state of object:

```
int GetValue() const { Counter++; return Value; } // compile error
```

To solve the problem cast the *this* pointer to non-const:

```
int GetValue() const
{
  ((Test *)this)->Counter++;
   return Value;
 }
```

or better

```
int GetValue() const
{
  (const_cast<Test *>(this))->Counter++;
   return Value;
 }
```

Casts (1)

The traditional explicit C cast (*new type) expression* is still in use:

```
double d = 5.6;
int i;
i = (int)d;
```

C++ has 4 new casting operators:

static_cast <*new type*> (*expression*)
dynamic_cast <*new type*> (*expression*)
reinterpret_cast <*new type*> (*expression*)
const_cast <*new type*> (*expression*)
Turn attention that the expression is always in parentheses.

The C-style cast is suitable for conversions between primitive data types. For conversions between pointers the C++ new casting operators are preferred.

Generally, the *static*, *reinterpret* ja *const* casts do the same as the C-style cast but allow more control over how the conversion should be performed. They are also easier to find in the source code.

Dynamic cast correctness is checked during run-time.

Casts (2)

The *static_cast* checks a bit more that C-style cast and is therefore more secure.

```
double d = 5.6;
int i;
i = static_cast<int>(d) // the same as i = (int)d;
class Base { ….. };
class Derived : public Base { ….. };
Derived *pd = new Derived;
Base *pb = pd; // implicit cast
pd = pb; // compile error, implicit cast not allowed
pd = (Derived *)pb; // legal, but also a possible source of run-time errors
pd = static_cast<Derived *>(pb); // legal and possible source of run-time errors
```

But

```
class Class1 {……};
class Class2 {……};
Class1 *pc1 = new Class1;
Class2 *pc2 = new Class2;
pc2 = (Class2 *)pc1; // legal, but also a source of run-time errors
pc2 = static_cast<Class2 *>(pc1); // compile error, static cast not allowed
```

The *static_cast* checks whether the pointer and pointee data types are compatible.

Casts (3)

The *reinterpret_cast* checks nothing and allows to cast a pointer to any other type of pointer (exactly as C-style cast):

```
class Class1 {……};
class Class2 {……};
Class1 *pc1 = new Class1;
Class2 *pc2 = new Class2;
pc2 = (Class2 *)pc1; // legal, but also a source of run-time errors
pc2 = reinterpret_cast<Class2 *>(pc1); // legal, but also a source of run-time errors
```

Using the *reinterpret_cast* instead of C-style cast the programmer emphasizes that he knows about the possible risks. If the program has crashed, places where *reinterpret_cast* (they are easy to find) is used are good start points for searching the bugs.

Casts (4)

The *const_cast* is used to convert a constant to non-constant. Example:

```
void alien (char *); // a third-party function we have to use
void fun (const char *p)
{  // our function, by specification its argument must be const char *
  ……….
  alien(p); // compile error
  alien((char *)p); // legal, but may crash if p points to a string constant
  alien(const_cast<char *>(p)); // legal , but may crash if p points to a string constant
  ……..
}
fun("I am John"); // crashes when function alien tries to change this text
```

Generally, if you try to change a value declared as *const*, the behavior is undefined but mostly the program crashes.

```
char *pc = new char[10];
strcpy(pc, "I am John");
const char *cpc = pc;
fun(cpc); // works because cpc points to memory field that is not constant
```

Casts (5)

The *const_cast* is safer because it can adjust the qualifier but not change the underlying type:

```
class Class1 {……};
class Class2 {……};
Class1 c1;
const Class1 *pc1 = &c1;
Class2 *pc2 = const_cast<Class2 *>(pc1); // compile error, pc1 is from different type
```

Casts (6)

The *dynamic_cast* provides pointers run-time check (not compile-time as the other casts) on casts within an inheritance hierarchy.

```
class Base
{
  virtual void base_fun(); // the hierarchy must contain at least one virtual method
   ……..
};
class Derived : public Base { ….. };
Derived *pd;
Base *pb = new Base;
pd = static_cast<Derived *>(pb); // legal and possible source of run-time errors
pd = dynamic_cast<Derived *>(pb); // no compile error but when the program
                                  // runs, the result is null-pointer
if (!pd)
{
   ……..
}
pb = dynamic_cast<Base *>(pd); // legal, no any errorrs
```

If the hierarchy does not contain virtual functions, a compile error will follow.

New variable types (1)

In C any variable of any type is interpreted as false if its value is zero and as true if its value is not zero. This is still correct in C++.
To improve the readability of code, preprocessor definitions like

```
#define TRUE  1
#define FALSE 0
```

are used. In C++ there is an additional built-in type: bool

```
bool b1 = true, b2 = false;
```

Actually, b1 is stored as integer 1 and b2 as integer 0. Boolean variables are implicitly (i.e. automatically) converted into integers and vice versa:

```
int i = b1; // i is now 1
b1 = 10; // b1 is now true
```

Examples of usage:

```
while (b1 == true) {……}
while (b1) {……}
while (!b2) {……}
bool fun()
{ …..
   return true;
}
```

New variable types (2)

Pointer that points to nothing has value 0:

```
char *p = 0;
```

Rather often:

```
#define NULL 0
char *p = NULL;

void fun(char *p) {…….}
void fun(int i) {…….}
```

Problem:

```
fun(0); // as 0 is an integer, always the second function is called
```

Solution:

```
fun(nullptr); // the first function is called
fun(0); // the second function is called
```

nullptr is introduced in C++ v 11. Advised to use instead 0 when working with pointers.

Enumerations (1)

Let us have:

```
#define ENGINEER 1
#define TEACHER 2
#define PENSIONER 3
#define ADMINISTRATOR 4
class Person
{
  ……..
  int Occupation;
  ……..
};
Person John;
John.SetOccupation(ENGINEER);
```

In this way we can avoid usage of strings, i.e. the memory allocation and releasing for them.

Enumerations (2)

The alternative is to use enumeration classes:

```
enum class enum_name { list_of_constants };
```

Example:

```
enum class Occupation { Engineer, Teacher, Pensioner, Administrator };
```

Now:

```
class Person
{
  ……….
  Occupation profession;
  ……..
  void SetProfession(Occupation oc) { profession = oc; }
  Occupation GetProfession() { return profession; }
};
Person John;
John.SetProfession(Ocupation::Administrator);
printf("%d\n", John.GetProfession()); // prints 3
Occupation oc = Occupation::Teacher; // variable of type Occupation
printf("%d\n", oc); // prints 1
```

Enumerations (3)

Each constant in enumeration has an associated with it integer. By default the first value is associated with 0, the second with 1, etc. But we can set our own values, for example:

```
enum class Occupation { Engineer = 10, Teacher, Pensioner, Administrator };
        // Teacher is now associated with 11, Pensioner with 12, etc.
enum class Occupation { Engineer, Teacher, Pensioner = 10, Administrator };
        // Engineer is now associated with 0, Teacher with 1,
        // Pensioner with 10, Administrator with 11
```

Remark: enumeration classes are defined in C++ version 11 and higher. C has simple enumerations like

```
enum Occupation { Engineer, Teacher, Pensioner, Administrator };
```

They work well in C but their usage in C++, however, may lead to problems.