SUBJECT- DATA STRUCTURE USING C++

C++ classes [UNIT –I]

A class in C++ is a user-defined type or data structure declared with any of the keywords `class`, `struct` or `union` (the first two are collectively referred to as non-union classes) that has data and functions (also called member variables and member functions) as its members whose access is governed by the three access specifiers *private*, *protected* or *public*. By default access to members of a C++ class declared with the keyword `class` is *private*. The private members are not accessible outside the class; they can be accessed only through member functions of the class. The public members form an interface to the class and are accessible outside the class.

Instances of a class data type are known as objects and can contain member variables, constants, member functions, and overloaded operators defined by the programmer.

Differences between a `struct` and a `class` in C++

In C++, a class defined with the `class` keyword has private members and base classes by default. A structure is a class defined with the `struct` keyword.[1] Its members and base classes are public by default. In practice, structs are typically reserved for data without functions. When deriving a struct from a class/struct, default access-specifier for a base class/struct is public. And when deriving a class, default access specifier is private.

Aggregate classes

An aggregate class is a class with no user-declared constructors, no private or protected non-static data members, no base classes, and no virtual functions.[2] Such a class can be initialized with a brace-enclosed comma-separated list of initializer-clauses.[3] The following code has the same semantics in both C and C++.

```
struct C {
  int a;
  double b;
};

struct D {
  int a;
  double b;
  C c;
};

// initialize an object of type C with an initializer-list
C c = {1, 2.0};
```

```
// D has a sub-aggregate of type C. In such cases initializer-clauses can be nested
D d = {10, 20.0, {1, 2.0}};
```

POD-structs

A POD-struct (Plain Old Data Structure) is a non-union aggregate class that has no non-static data members of type non-POD-struct, non-POD-union (or array of such types) or reference, and has no user-defined assignment operator and no user-defined destructor.[1] A POD-struct could be said to be the C++ equivalent of a C `struct`. In most cases, a POD-struct will have the same memory layout as a corresponding struct declared in C.[4] For this reason, POD-structs are sometimes colloquially referred to as "C-style structs".[5]

Properties shared between structs in C and POD-structs in C++

- Data members are allocated so that later members have higher addresses within an object, except where separated by an access-specifier.[6]
- Two POD-struct types are layout-compatible if they have the same number of nonstatic data members, and corresponding nonstatic data members (in order) have layout-compatible types.[7]
- A POD-struct may contain unnamed padding.[8]
- A pointer to a POD-struct object, suitably converted using a reinterpret cast, points to its initial member and vice versa, implying that there is no padding at the beginning of a POD-struct.[8]
- A POD-struct may be used with the offsetof macro.[9]

Declaration and usage

C++ classes have their own members. These members include variables (including other structures and classes), functions (specific identifiers or overloaded operators) known as member functions, constructors and destructors. Members are declared to be either publicly or privately accessible using the `public:` and `private:` access specifiers respectively. Any member encountered after a specifier will have the associated access until another specifier is encountered. There is also inheritance between classes which can make use of the `protected:` specifier.

Global and local class

[edit]

A class defined outside all functions is a global class because its objects can be created from anywhere in the program. If it is defined within a function body then it's a local class because objects of such a class are local to the function scope.

Basic declaration and member variables

Non-union classes are declared with the `class` or `struct` keyword. Declaration of members are placed within this declaration.

```
struct Person {
```

```
class Person {
```

```
  string name;
  int age;
};
```

```
public:
  string name;
  int age;
};
```

The above definitions are functionally equivalent. Either code will define objects of type `Person` as having two public data members, `name` and `age`. The semicolons after the closing braces are mandatory.

After one of these declarations (but not both), `Person` can be used as follows to create newly defined variables of the `Person` datatype:

```
#include <iostream>
#include <string>

struct Person {
  std::string name;
  int age;
};

int main() {
  Person a;
  Person b;
  a.name = "Calvin";
  b.name = "Hobbes";
  a.age = 30;
  b.age = 20;
  std::cout << a.name << ": " << a.age << std::endl;
  std::cout << b.name << ": " << b.age << std::endl;
}
```

Executing the above code will output

```
Calvin: 30
Hobbes: 20
```

Member functions

[

An important feature of the C++ class are **member functions**. Each datatype can have its own built-in functions (referred to as member functions) that have access to all (public and private) members of the datatype. In the body of these non-static member functions, the keyword `this` can be used to refer to the object for which the function is called. This is commonly implemented by passing the address of the object as an implicit first argument to the function.[10] Take the above `Person` type as an example again:

```
#include <iostream>

class Person {
 public:
  void Print() const;

 private:
  std::string name_;
```

```
  int age_=5; //C++ 11
};

void Person::Print() const {
  std::cout << name_ << ':' << age_ << '\n';
  // "name_" and "age_" are the member variables. The "this" keyword is an
  // expression whose value is the address of the object for which the member
  // was invoked. Its type is "const Person*", because the function is
declared
  // const.
}
```

In the above example the `Print` function is declared in the body of the class and defined by qualifying it with the name of the class followed by `::`. Both `name_` and `age_` are private (default for class) and `Print` is declared as public which is necessary if it is to be used from outside the class.

With the member function `Print`, printing can be simplified into:

```
a.Print();
b.Print();
```

where `a` and `b` above are called senders, and each of them will refer to their own member variables when the `Print()` function is executed.

It is common practice to separate the class or structure declaration (called its interface) and the definition (called its implementation) into separate units. The interface, needed by the user, is kept in a header and the implementation is kept separately in either source or compiled form.

Inheritance

The layout of non-POD classes in memory is not specified by the C++ standard. For example, many popular C++ compilers implement single inheritance by concatenation of the parent class fields with the child class fields, but this is not required by the standard. This choice of layout makes referring to a derived class via a pointer to the parent class type a trivial operation.

For example, consider

```
struct P {
  int x;
};
struct C : P {
  int y;
};
```

An instance of `P` with a `P* p` pointing to it might look like this in memory:

```
┌──────┐
│P::x│
└──────┘

↑
```

p

An instance of C with a P* p pointing to it might look like this:

↑
p

Therefore, any code that manipulates the fields of a P object can manipulate the P fields inside the C object without having to consider anything about the definition of C's fields. A properly written C++ program shouldn't make any assumptions about the layout of inherited fields, in any case. Using the static_cast or dynamic_cast type conversion operators will ensure that pointers are properly converted from one type to another.

Multiple inheritance is not as simple. If a class D inherits P and C, then the fields of both parents need to be stored in some order, but (at most) only one of the parent classes can be located at the front of the derived class. Whenever the compiler needs to convert a pointer from the D type to either P or C, the compiler will provide an automatic conversion from the address of the derived class to the address of the base class fields (typically, this is a simple offset calculation).

For more on multiple inheritance, see virtual inheritance.

The final keyword limits the ways in which a class can be subclassed.[11] Subclasses of a class are prevented from overriding methods marked as final by the parent class.[12][13] Final classes cannot be inherited.[13] This allows *devirtualization*, the removal of the use of vtables for method lookup, thus allowing the inlining of method calls on final classes.[14][15]

final is not a reserved word in C++, and is instead defined as a contextual keyword, in order to not conflict with uses of the identifier 'final' in existing codebases.[16][17]

Overloaded operators

: Operators in C and C++

In C++, operators, such as + - * /, can be overloaded to suit the needs of programmers. These operators are called **overloadable operators**.

By convention, overloaded operators should behave nearly the same as they do in built-in datatypes (int, float, etc.), but this is not required. One can declare a structure called Integer in which the variable *really* stores an integer, but by calling Integer * Integer the sum, instead of the product, of the integers might be returned:

```
struct Integer {
  Integer() = default;
  Integer(int j) : i{j} {}
```

```
  Integer operator*(const Integer& k) const {
    return Integer(i + k.i);
  }

  int i = 0;
};
```

The code above made use of a constructor to "construct" the return value. For clearer presentation (although this could decrease efficiency of the program if the compiler cannot optimize the statement into the equivalent one above), the above code can be rewritten as:

```
Integer operator*(const Integer& k) const {
  Integer m;
  m.i = i + k.i;
  return m;
}
```

Programmers can also put a prototype of the operator in the `struct` declaration and define the function of the operator in the global scope:

```
struct Integer {
  Integer() = default;
  Integer(int j) : i{j} {}

  Integer operator*(const Integer& k) const;

  int i = 0;
};

Integer Integer::operator*(const Integer& k) const {
  return Integer(i * k.i);
}
```

`i` above represents the sender's own member variable, while `k.i` represents the member variable from the argument variable `k`.

The `const` keyword appears twice in the above code. The first occurrence, the argument `const integer& k`, indicated that the argument variable will not be changed by the function. The second incidence at the end of the declaration promises the compiler that the sender would not be changed by the function run.

In `const integer& k`, the ampersand (&) means "pass by reference". When the function is called, a reference to the variable will be passed to the function, rather than the value of the variable.

Note that arity, associativity and precedence of operators cannot be changed.

Binary overloadable operators

Binary operators (operators with two arguments) are overloaded by declaring a function with an "identifier" *operator (something)* which calls one single argument. The variable on the left of the operator is the sender while that on the right is the argument.

```
Integer i = 1;
/* we can initialize a structure variable this way as
   if calling a constructor with only the first
   argument specified. */
Integer j = 3;
/* variable names are independent of the names of the
   member variables of the structure. */
Integer k = i * j;
std::cout << k.i << '\n';
```

'3' would be printed.

The following is a list of binary overloadable operators:

Operator	General usage
+ - * / %	Arithmetic calculation
^ & ! << >>	Bitwise calculation
< > == != <= >=	Logical comparison
&&	Logical conjunction
||	Logical disjunction
= <<= >>=	Compound assignment
,	(no general usage)

The '=' (assignment) operator between two variables of the same structure type is overloaded by default to copy the entire content of the variables from one to another. It can be overwritten with something else, if necessary.

Operators must be overloaded one by one, in other words, no overloading is associated with one another. For example, < is not necessarily the opposite of >.

Unary overloadable operators

While some operators, as specified above, takes two terms, sender on the left and the argument on the right, some operators have only one argument - the sender, and they are said to be "unary". Examples are the negative sign (when nothing is put on the left of it) and the "logical NOT !).

Sender of unary operators may be on the left or on the right of the operator. The following is a list of unary overloadable operators:

Operator	General usage	Position of sender
+ -	Positive / negative sign	right
* &	Dereference	right
! ~	Logical / bitwise NOT	right
++ --	Pre-increment / decrement	right
++ --	Post-increment / decrement	left

The syntax of an overloading of a unary operator, where the sender is on the right, is as follows:

```
return_type operator@ ()
```

When the sender is on the left, the declaration is:

```
return_type operator@ (int)
```

`@` above stands for the operator to be overloaded. Replace `return_type` with the datatype of the return value (`int`, `bool`, structures etc.)

The `int` parameter essentially means nothing but a convention to show that the sender is on the left of the operator.

`const` arguments can be added to the end of the declaration if applicable.

The square bracket `[]` and the round bracket `()` can be overloaded in C++ classes. The square bracket must contain exactly one argument, while the round bracket can contain any specific number of arguments, or no arguments.

The following declaration overloads the square bracket.

```
return_type operator[] (argument)
```

The content inside the bracket is specified in the `argument` part.

Round bracket is overloaded a similar way.

```
return_type operator() (arg1, arg2, ...)
```

Contents of the bracket in the operator call are specified in the second bracket.

In addition to the operators specified above, the arrow operator (`->`), the starred arrow (`->*`), the `new` keyword and the `delete` keyword can also be overloaded. These memory-or-pointer-related operators must process

memory-allocating functions after overloading. Like the assignment (=) operator, they are also overloaded by default if no specific declaration is made.

Constructors

Sometimes programmers may want their variables to take a default or specific value upon declaration. This can be done by declaring constructors.

```
Person::Person(string name, int age) {
  name_ = name;
  age_ = age;
}
```

Member variables can be initialized in an initializer list, with utilization of a colon, as in the example below. This differs from the above in that it initializes (using the constructor), rather than using the assignment operator. This is more efficient for class types, since it just needs to be constructed directly; whereas with assignment, they must be first initialized using the default constructor, and then assigned a different value. Also some types (like references and `const` types) cannot be assigned to and therefore must be initialized in the initializer list.

```
Person(std::string name, int age) : name_(name), age_(age) {}
```

Note that the curly braces cannot be omitted, even if empty.

Default values can be given to the last arguments to help initializing default values.

```
Person(std::string name = "", int age = 0) : name_(name),
age_(age) {}
```

When no arguments are given to the constructor in the example above, it is equivalent to calling the following constructor with no arguments (a default constructor):

```
Person() : name_(""), age_(0) {}
```

The declaration of a constructor looks like a function with the same name as the datatype. In fact, a call to a constructor can take the form of a function call. In that case an initialized `Person` type variable can be thought of as the return value:

```
int main() {
  Person r = Person("Wales", 40);
  r.Print();
}
```

An alternate syntax that does the same thing as the above example is

```
int main() {
```

```
  Person r("Wales", 40);
  r.Print();
}
```

Specific program actions, which may or may not relate to the variable, can be added as part of the constructor.

```
Person() {
  std::cout << "Hello!" << std::endl;
}
```

With the above constructor, a "Hello!" will be printed when the default `Person` constructor is invoked.

Default constructor

Default constructors are called when constructors are not defined for the classes.

```
struct A {
  int b;
};
// Object created using parentheses.
A* a = new A();  // Calls default constructor, and b will be
initialized with '0'.
// Object created using no parentheses.
A* a = new A;  // Allocate memory, then call default constructor,
and b will have value '0'.
// Object creation without new.
A a;  // Reserve space for a on the stack, and b will have an
unknown garbage value.
```

However, if a *user defined constructor* was defined for the class, both of the above declarations will call this user defined constructor, whose defined code will be executed, but no default values will be assigned to the variable b.

Destructors

A destructor is the inverse of a constructor. It is called when an instance of a class is destroyed, e.g. when an object of a class created in a block (set of curly braces "{}") is deleted after the closing brace, then the destructor is called automatically. It will be called upon emptying of the memory location storing the variables. Destructors can be used to release resources, such as heap-allocated memory and opened files when an instance of that class is destroyed.

The syntax for declaring a destructor is similar to that of a constructor. There is no return value and the name of the function is the same as the name of the class with a tilde (~) in front.

```
~Person() {
  std::cout << "I'm deleting " << name_ << " with age " << age_
<< std::endl;
```

```
}
```

Similarities between constructors and destructors

- Both have same name as the class in which they are declared.
- If not declared by user both are available in a class by default but they now can only allocate and deallocate memory from the objects of a class when an object is declared or deleted.
- For a derived class: During the runtime of the base class constructor, the derived class constructor has not yet been called; during the runtime of the base class destructor, the derived class destructor has already been called. In both cases, the derived class member variables are in an invalid state.

Class templates

: Template (programming)

In C++, class declarations can be generated from class templates. Such class templates represent a family of classes. An actual class declaration is obtained by *instantiating* the template with one or more template arguments. A template instantiated with a particular set of arguments is called a template specialization.

Properties

The syntax of C++ tries to make every aspect of a class look like that of the basic datatypes. Therefore, overloaded operators allow classes to be manipulated just like integers and floating-point numbers, arrays of classes can be declared with the square-bracket syntax (`some_structure variable_name[size]`), and pointers to classes can be dereferenced in the same way as pointers to built-in datatypes.

Memory consumption

[edit]

The memory consumption of a structure is at least the sum of the memory sizes of constituent variables. Take the `TwoNums` structure below as an example.

```
struct TwoNums {
  int a;
  int b;
};
```

The structure consists of two integers. In many current C++ compilers, integers are 32-bit integers by default, so each of the member variables consume four bytes of memory. The entire structure, therefore, consumes at least (or exactly) eight bytes of memory, as follows.

```
+----+----+
| a  | b  |
+----+----+
```

However, the compiler may add padding between the variables or at the end of the structure to ensure proper data alignment for a given computer architecture, often padding variables to be 32-bit aligned. For example, the structure

```
struct BytesAndSuch {
  char c;
  char C;
  char D;
  short int s;
  int i;
  double d;
};
```

could look like

```
+-+-+-+-+--+--+----+--------+
|c|C|D|X|s |XX|  i |   d    |
+-+-+-+-+--+--+----+--------+
```

in memory, where `X` represents padded bytes based on 4 bytes alignment.

As structures may make use of pointers and arrays to declare and initialize its member variables, memory consumption of structures is not necessarily constant. Another example of non-constant memory size is template structures.

Bit fields

[edit]

Bit fields are used to define the class members that can occupy less storage than an integral type. This field is applicable only for integral types (int, char, short, long, etc.) and enumeration types (e.g. std::byte) and excludes float or double.

```
struct A {
  unsigned a:2;  // Possible values 0..3, occupies first 2 bits
of int
  unsigned b:3;  // Possible values 0..7, occupies next 3 bits of
int
  unsigned :0;  // Moves to end of next integral type
  unsigned c:2;
  unsigned :4;  // Pads 4 bits in between c & d
  unsigned d:1;
  unsigned e:3;
};
```

- Memory structure

```
4 byte int  4 byte int
```

```
   [1][2][3][4][5][6][7][8]
   [1]                        [2]                        [3]
[4]
   [a][a][b][b][b][ ][ ][ ] [ ][ ][ ][ ][ ][ ][ ] [ ][ ][ ][
][ ][ ][ ][ ] [ ][ ][ ][ ][ ][ ][ ]

   [5]                        [6]                        [7]
[8]
   [c][c][ ][ ][ ][ ][d][e] [e][e][ ][ ][ ][ ][ ] [ ][ ][ ][
][ ][ ][ ][ ] [ ][ ][ ][ ][ ][ ][ ]
```

Unions are also allowed to have bit-field members:

```
union A {
  unsigned a:2;
  unsigned b:3;
  unsigned :0; // Does nothing
  unsigned c:2;
  unsigned :4; // Does nothing in this case; if the bit-field's
width were large enough, it would change the union's size to fit
```