C++ classes

From Wikipedia, the free encyclopedia
(Redirected from C language struct)

The C++ programming language allows programmers to separate program-specific datatypes through the use of **classes**. Instances of these datatypes are known as objects and can contain member variables, constants, member functions, and overloaded operators defined by the programmer. Syntactically, classes are extensions of the C struct, which cannot contain functions or overloaded operators.

Contents

- 1 Differences between struct and classes in C++
  - 1.1 Aggregate classes
  - 1.2 POD-structs
    - 1.2.1 Properties shared between structs in C and POD-structs in C++
- 2 Declaration and usage
  - 2.1 Basic declaration and member variables
  - 2.2 Member functions
  - 2.3 Inheritance
  - 2.4 Overloaded operators
    - 2.4.1 Binary overloadable operators
    - 2.4.2 Unary overloadable operators
    - 2.4.3 Overloading brackets
  - 2.5 Constructors
  - 2.6 Destructors
- 3 Class templates
- 4 Properties
  - 4.1 Memory consumption
  - 4.2 Pass by reference
  - 4.3 The this keyword
- 5 See also
- 6 References

Differences between struct and classes in C++

In C++, a *structure* is a class defined with the `struct` keyword.[1] Its members and base classes are public by default. A class defined with the `class` keyword has private members and base classes by default. This is the only difference between structs and classes in C++.

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};

// D has a sub-aggregate of type C. In such cases initializer-clauses can be nested
D d = {10, 20, {1, 2}};
```

POD-structs

A POD-struct (Plain Old Data Structure) is an 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 methods, 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.

Basic declaration and member variables

Classes are declared with the `class` or `struct` keyword. Declaration of members are placed within this declaration.

```
struct person
{
  string name;
  int age;
};
struct people
{
  int a;
  float b;
  person p;
};
```

```
class person
{
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>
using namespace std;

class person
{
public:
  string name;
  int age;
};

int main()
{
  person a, b;
  a.name = "Calvin";
  b.name = "Hobbes";
  a.age = 30;
  b.age = 20;
  cout << a.name << ": " << a.age << endl;
  cout << b.name << ": " << b.age << endl;
  return 0;
}
```

Executing the above code will output

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

Member functions

An important feature of the C++ class and structure are **member functions**. Each datatype can have its own built-in functions (referred to as methods) 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:

```
class person
{
  std::string name;
  int age;
public:
  person() : age(5) { }
  void print() const;
};

void person::print() const
{
  cout << name << ";" << age << endl;
  /* "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

```
class P
{
    int x;
};
class C : public 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:

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 `P1` and `P2`, 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 `P1` or `P2`, 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.

Overloaded operators

For more details on this topic, see 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
{
    int i;
    integer(int j = 0) : i(j) {}
    integer operator*(const integer &k) const
    {
        return integer (i + k.i);
    }
};
```

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
{
    int i;

    integer(int j = 0) : i(j) {}

    integer operator*(const integer &k) const;
};

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 pointer to the variable will be passed to the function, rather than the value of the variable.

The same overloading properties above apply also to classes.

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;
cout << k.i << endl;
```

'4' would be printed.

The following is a list of binary overloadable operators:

Operator	General usage
+ - * / %	Arithmetic calculation
<< >>	Bitwise calculation
< > == != <= >=	Logical comparison
&&	Logical conjunction
= <<= >>=	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" (exclamation mark, !).

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.

Overloading brackets

The square bracket `[]` and the round bracket `()` can be overloaded in C++ structures. 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 software engineers may want their variables to take a default or specific value upon declaration. This can be done by declaring constructors.

```
person(string N, int A)
{
    name = N;
    age = A;
}
```

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 N, int A) : name(N), age(A) {}
```

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 N = "", int A = 0) : name(N), age(A) {}
```

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 name as the same as the datatype. In fact, we can really call the constructor in form of a function call. In that case a `person` type variable would be the return value:

```
int main()
{
    person r = person("Wales", 40);
    r.print();
}
```

The above code creates a temporary person object, and then assigns it to `r` using the copy constructor. A better way of creating the object (without unnecessary copying) 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!" << endl;
}
```

With the above constructor, a "Hello!" will be printed in case a `person` variable with no specific value is initialized.

Destructors

A destructor is the reverse 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 variable. 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 method is the same as the name of the class with a tilde (~) in front.

```
~person()
{
    cout << "I'm deleting " << name << " with age " << age << endl;
}
```

Class templates

Main article: 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 structure look like that of the basic datatypes. Therefore, overloaded operators allow structures to be manipulated just like integers and floating-point numbers, arrays of structures can be declared with the square-bracket syntax (`some_structure variable_name[size]`), and pointers to structures can be dereferenced in the same way as pointers to built-in datatypes.

Memory consumption

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 bytes_and_such
{
   char c;
   char C;
   short int s;
   int i;
   double d;
};
```

could look like

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

in memory, where XX is two unused bytes.

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.

Pass by reference

For more details on this topic, see evaluation strategy.

Many programmers prefer to use the ampersand (&) to declare the arguments of a function involving structures. This is because by using the dereferencing ampersand only one word (typically 4 bytes on a 32 bit machine, 8 bytes on a 64 bit machine) is required to be passed into the function, namely the memory location to the variable. Otherwise, if pass-by-value is used, the argument needs to be copied every time the function is called, which is costly with large structures.

Since pass-by-reference exposes the original structure to be modified by the function, the `const` keyword should be used to guarantee that the function does not modify the parameter (see const-correctness), when this is not intended.

The *this* keyword

Main article: this (computer science)

To facilitate structures' ability to reference themselves, C++ implements the `this` keyword for all member functions. The `this` keyword acts as a pointer to the current object.[11]. Its type is that of a pointer to the current object.

The `this` keyword is especially important for member functions with the structure itself as the return value:

```
complex& operator+=(const complex & c)
{
    realPart += c.realPart;
    imagPart += c.imagPart;
    return *this;
}
```

As stated above, `this` is a pointer, so the use of the asterisk (*) is necessary to convert it into a reference to be returned.

See also

- Object composition
- Virtual inheritance
- Type conversion

References

1. ^ **a** **b** ISO/IEC (2003). *ISO/IEC 14882:2003(E): Programming Languages - C++ §9 Classes [class]* para. 4
2. **^** ISO/IEC (2003). *ISO/IEC 14882:2003(E): Programming Languages - C++ §8.5.1 Aggregates [dcl.init.aggr]* para. 1
3. **^** ISO/IEC (2003). *ISO/IEC 14882:2003(E): Programming Languages - C++ §8.5.1 Aggregates [dcl.init.aggr]* para. 2
4. **^** "What's this "POD" thing in C++ I keep hearing about?" (http://www.comeaucomputing.com/techtalk/#pod) . Comeau Computing. http://www.comeaucomputing.com/techtalk/#pod. Retrieved 2009-01-20.
5. **^** Henricson, Mats; Nyquist, Erik (1997). *Industrial Strength C++*. Prentice Hall. ISBN 0-13-120965-5.
6. **^** ISO/IEC (2003). *ISO/IEC 14882:2003(E): Programming Languages - C++ §9.2 Class members [class.mem]* para. 12
7. **^** ISO/IEC (2003). *ISO/IEC 14882:2003(E): Programming Languages - C++ §9.2 Class members*

[class.mem] para. 14
8. ^ **a b** ISO/IEC (2003). *ISO/IEC 14882:2003(E): Programming Languages - C++ §9.2 Class members [class.mem]* para. 17
9. **^** ISO/IEC (2003). *ISO/IEC 14882:2003(E): Programming Languages - C++ §18.1 Types [lib.support.types]* para. 5
10. **^** "thiscall (C++)" (http://msdn.microsoft.com/en-us/library/ek8tkfbw(VS.71).aspx) . http://msdn.microsoft.com/en-us/library/ek8tkfbw(VS.71).aspx. Retrieved 2009-01-26.
11. **^** "this" (http://www.cppreference.com/wiki/keywords/this) . *C++ Reference*. http://www.cppreference.com/wiki/keywords/this.

General References:

- Cplusplus.com tutorial lesson 5.2 (http://www.cplusplus.com/doc/tutorial/classes2.html) , accessed in January 2006
- Chan, S. "Tutorial on C++ and STL", Hong Kong, 2006
- Cplusplus.com tutorial lesson 2.5 (http://www.cplusplus.com/doc/tutorial/basic_io.html) , accessed in February 2006

Retrieved from "http://en.wikipedia.org/wiki/C%2B%2B_classes"
Categories: C++ | Articles with example C++ code

- This page was last modified on 10 March 2011 at 10:06.
- Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. See Terms of Use for details.
  Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.