In this article, Bjarne Stroustrup describes C++, the superset of C he created.

Object-Oriented Programming

Concepts do not usually come as self-contained entities. On the contrary, most concepts relate to other concepts in a variety of ways. For example, the concepts of airplane and car relate to those of vehicle and transport; the concepts of mammal and bird relate to each other through the more general concept of vertebrate animal, through the concept of food, and so forth; and the concepts of a circle, rectangle, and polygon involve the general concept of a shape.

Therefore, representing concepts directly as types in a program also requires ways of expressing the relations between types. C++ lets you specify hierarchically organized classes. This is the key feature supporting object-oriented programming. Hierarchical organization is an extremely important way of coping with complex issues in many fields and has, not surprisingly, also proven to be a good way of organizing programs in a wide variety of application areas.

Consider defining a type shape for use in a graphics system. The system has to support circles, triangles, squares, and many other shapes. First, you specify a class that defines the general properties of all shapes:

```cpp
class shape {
  point center;
  color col;
  //...
  public:
  point where() { return center; }
  void move (point to)
  { center = to; draw(); }
  virtual void draw();
  virtual void rotate(int);
  //...
};
```

You can define the calling interfaces for `draw()` and `rotate()`, but you cannot yet define their implementation. They are, therefore, declared virtual (the Simula67 and C++ term for “to be defined later in a class derived from this one”). They will be defined for each specific shape. Given this definition of class shape, you can write general functions manipulating shapes:

```cpp
void rotate_all(shape* v[], int size, int angle)
// rotate all members of
// vector "v" of size "size"
// "angle" degrees
{
  for (int i = 0; i < size; i++)
    v[i]->rotate(angle);
}
```

For each shape `v[i]`, the proper `rotate()` function for the actual type of the object will be called. That “actual type” is not known at compile time.

To define a particular shape, you must say that it is a shape and specify its particular properties:
A class is said to be derived from another class, which is then called its base class. Here, circle is derived from shape, and shape has a base class of circle. A derived class is said to inherit the properties of its base. In addition to such inherited properties, a derived class has its own specific properties. For example, class circle has the member radius in addition to the members col and center that it inherited from class shape.

Note that the new shape center was added without modifying "old code," such as the rotate-all() function and other shapes. The ability to extend a program by adding new variations of a basic concept (i.e., adding new derived classes given a base class) without touching old code is a major boon. Using traditional techniques, such additions require access to the source code of the system you want to extend, require understanding of the key implementation details of the old code, and carry the risk of introducing errors in the already-tested old code. Furthermore, using derived classes, improvements and bug fixes done to a base class are automatically "inherited" by every class derived from it.

I chose the "shape" example because everyone understands about shapes, not because object-oriented programming has anything particular to do with graphics. Graphics is a good area for object-oriented techniques, but most uses of such techniques in C++ have nothing to do with graphics. Other examples are compilers, operating-system kernels and device drivers, switching software, and network simulations.

In many contexts, it is important that the C++ virtual-function mechanism be nearly as efficient as a "normal" function call. The additional run-time overhead is about five memory references (depending on the machine architecture and the compiler), and the memory overhead is one word per object plus one word per virtual function per class.

C++ provides multiple inheritance (see reference 7), or the ability to derive a class from more than one direct base class. For example, if you have a class task representing the concept of a concurrent activity, and a class displayed representing the concept of something displayed on the screen, you might write:

```cpp
class displayed_task
    public displayed, public task
{...
```

Now a displayed_task is really both a displayed and a task, so a displayed_task can be used wherever a displayed or a task is required:

```cpp
void wait(task*,int);
// do something to a task
void update(displayed*);
// do something to a displayed
f()
{
    // make a displayed_task:
displayed_task* dtp =
        new displayed_task {
            /* appropriate arguments */ } ;
    wait(dtp,10);
    // use displayed-task as a task
    update(dtp);
    // displayed-task as-displayed
}
```

Naturally, the usual type-checking rules, ambiguity rules, and encapsulation mechanisms are applied to multiple inheritance to ensure the usual degree of safety and efficiency.
Why C++?

What distinguishes C++ from other programming languages? C++ was designed under severe constraints of compatibility, internal consistency, and efficiency. No feature was included that would cause a serious incompatibility with C at the source or linker levels; would cause run-time or space overheads for a program that did not use it; would increase run time or requirements for a C program; would significantly increase the compile time compared with C; or could only be implemented by making more demands than in a traditional programming environment.

Traditional languages such as C, FORTRAN, Pascal, and Modula-2 don't provide anything comparable to C++'s features for data abstraction and object-oriented programming. This gives the C++ programmer a strong advantage when it comes to understanding, writing, and maintaining programs. It's often important that the improved structure of C++ programs be achieved without sacrificing efficiency or restricting the range of areas for which the language is suitable.

Ada provides facilities for data abstraction that may not be as elegant as C++'s but should be about as effective in actual use. But Ada doesn't provide an inheritance mechanism to support object-oriented programming, so C++ has greater expressive power in this area.

C++ is distinguished among languages that support object-oriented programming, such as Smalltalk, by a variety of factors: its emphasis on program structure; the flexibility of encapsulation mechanisms; its smooth support of a range of programming paradigms; the portability of C++ implementations; the run-time efficiency (in both time and space) of C++ code; and its ability to run without a large run-time system.

C++ is a programming language in the traditional sense and is not a complete program development system or a complete execution environment. It can be installed easily into an existing C program development or execution environment, and C++-specific tools can then be added as needed. In addition, several C++-specific environments are being built to suit specific needs (see references 8 and 9).

The emphasis on explicit static structure (as opposed to a weak type-checking, as in C, or purely dynamic typechecking, as in Smalltalk) is particularly important for projects involving many programmers and for individual programmers using large libraries written by others. C++’s strong type-checking and encapsulation mechanisms have repeatedly proven themselves by dramatically reducing integration time for larger projects. Similarly, C++ provides a good base for designing libraries with precisely defined, elegant, and statically checked interfaces.

C++ has a single, very flexible, type system. This makes it possible to use hybrid programming styles without violating the C++ type system. It also lets you choose a style of programming closely matching individual application areas.

References


