A C++ Tutorial

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This is a tutorial introduction to the C++ programming language. With few exceptions C++ is a superset of the C programming language. After the introduction, about a third of the text presents the more conventional features of C++: basic types, declarations, expressions, statements, and functions. The remainder concentrates on C++'s facilities for data abstraction: user-defined types, data-hiding, user-defined operators, and hierarchies of user-defined types. Finally there are a few comments on program structure, compatibility with C, efficiency and a caveat.

September 10, 1984

A C++ Tutorial

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1 Introduction

"The only way to learn a new programming language is by writing programs in it" (K&R[1],page 5).

This tutorial will guide you through a sequence of C++ programs and program fragments. At the end you should have a general idea about the facilities of C++, and enough information to write simple programs. Little is assumed about your knowledge of programming, but the progress through the concepts may be mind-boggling if you are a novice. If you are familiar with C you will notice that with few exceptions C++ is a superset of it. However, the examples have been chosen so that only few could have been written identically in C.

A precise and complete explanation of the concepts involved in even the smallest complete example would require pages of definitions. To avoid this paper turning into a manual or a discussion of general ideas, examples are presented first with only the briefest definition of the terms used. Many of these terms are reviewed later when a larger body of examples are available to aid the discussion. References 2 and 3 contain more systematic and complete discussions of C++.

Output

Let us first of all write a program to write a line of output:

```
#include <stream.h>

main()
{
    cout<<"Hello, world\n";
}
```

The line `#include <stream.h>` instructs the compiler to "include" the declarations of the standard input and output facilities into the program. Without these declarations the statement `cout<<"Hello, world\n";` would make no sense. The operator `<<` ("put to") writes its second argument onto its first (in this case, the string `"Hello, world\n"` onto the standard output stream `cout`). A string is a sequence of characters surrounded by double quotes; in a string the backslash character `\` followed by another character denotes a single "special" character; in this case `\n` is the newline character, so that the characters written are `Hello, world` and newline.

The rest of the program

```
main() { ... }
```

defines a function called `main`. A program must have a function named `main`, and the program is started by executing that function.

Compilation

Where did the output stream `cout` and the code implementing the output operator `<<` come from? A C++ program must be compiled to produce executable code (the compilation process is essentially the same as for C, and shares most of the programs involved): The program text is read and analyzed, and if no error is found code is generated. Then the program is examined to find names and operators that have been used but not defined (in our case `cout` and `<<`). If possible, the program is then completed by adding the missing definitions from a library (there is a standard library and users can provide their own). In our case `cout` and `<<` were declared (in `stream.h`); that is, their types were given, but no details of their implementation were provided. The standard library contains the specification of the space and initialization code for `cout` and the code for `<<`. Naturally there are many other things in that library, some of which are declared in `stream.h`, but only the subset of the library needed to complete our program is added to the compiled version. The C++ compile command is typically called `CC`. It is used like `cc` for C programs; see your manual for details.

Input

The following (rather verbose) conversion program prompts you to enter a number of inches. When you have done that it will print the corresponding number of centimeters.

```
#include <stream.h>

main()
{
    int inch;
    cout<<"inches=";
    cin>>inch;
    cout<<inch;
    cout<<" in = ";
    cout<<inch*2.54;
    cout<<" cm\n";
}
```

The first line of `main()` declares an integer variable `inch`. Its value is read in using the operator `>>` ("get from") on the standard input stream `cin`. The declarations of `cin` and `>>` are of course found in `<stream.h>`.

After executing it your terminal might look like this

```
inches=12
12 in = 30.48 cm
```

This example had one statement per output operation; this is unnecessarily verbose. The output operator `<<` can be applied to its own result, so that the last four output operations could have been written in a single statement:

```
cout<<inch<<" in = "<<inch*2.54<<" cm\n";
```

Input and output will be described in greater detail below. In fact, this whole tutorial can be seen as an explanation of how it is possible to write the programs above in a language that does not provide an input or an output operator! That is, the C++ language as described in the reference manual[2] does not define facilities for input and output; instead, the operators `>>` and `<<` were defined using only language facilities available to every programmer.

2 Types and Declarations

Every name and every expression has a type that determines the operations that may be performed on it. For example, the declaration

```
int inch;
```

specifies that `inch` is of type `int`; that is, `inch` is an integer variable.

A declaration is a statement that introduces a name into the program. It must specify a type for that name. A type defines the proper use of a name or an expression. Operators like + - * and / are defined for integers; so are, after `stream.h` has been included, the input operator `>>` and the output operator `<<`.

The type of an object determines not only which operations can be applied to it, but also the meaning of those operations. For example, the statement

```
        cout<<inch<<" in = "<<inch*2.54<<" cm\n";
```

correctly treats the 4 values to be written out differently. The strings are printed as presented, whereas the integer `inch` is converted from its internal representation to a character representation fit for human eyes. So is the floating point number obtained by multiplying the integer `inch` by the floating point constant `2.54`.

C++ has several basic types and several ways of creating new ones. The simplest forms of C++ types are presented in the sections below; the more interesting ones are saved for later.

Basic Types

The basic types, corresponding most directly to hardware storage facilities are:

```
        char   short   int   long
        float double
```

The first four are used for representing integers. A variable of type `char` is of the natural size to hold a character on a given machine (typically a byte), and a variable `int` is of the natural size for integer arithmetic on a given machine (typically a word). The range of integers that can be represented by a type depends on its size. In C++ "sizes" are measured in multiples of the size of a `char`, so by definition `char` has size one. The relation between the integer types can be written like this:

1=sizeof(char)≤sizeof(short)≤sizeof(int)≤sizeof(long)

In general it is foolish to assume more about the sizes of integers. In particular, it is not true for all machines that an integer is large enough to hold a pointer.

`Float` and `double` are used for representing floating point numbers.

sizeof(float)≤sizeof(double)

The adjective `const` can be applied to a basic type to yield a type that has identical properties to the original, except that the value of variables of a `const` type cannot be changed after initialization.

```
        const float pi = 3.14;
        const char plus = '+';
```

Note that most often a constant defined like this need not occupy storage; its value can simply be used directly where needed. A constant MUST be initialized at the point of declaration, as shown above. For variables the initialization is optional, but strongly recommended. There are very few good reasons for introducing a local variable without initializing it.

The arithmetic operators

```
        +
        -   (both unary and binary)
        *
        /
```

can be used for any combination of these types. So can the comparison operators

```
        ==    (equal)
        !=    (not equal)
        <     (less than)
        >     (greater than)
        <=    (less than or equal)
        >=    (greater than or equal)
```

though if you use `==` or `!=` on the result of floating point computations you are likely to get what you deserve. Note that integer division will yield an integer result: `7/2` is `3`. The operator `%` can be used on integers to produce the remainder: `7%2` is `1`.

In assignments and in arithmetic operations C++ will perform all meaningful conversions between the basic types so that they can be mixed freely. For example:

```
double d = 1;
int i = 1;
d = d+i;
i = d+i;
```

The compiler will, however, warn about loss of precision in the last assignment.

Derived Types

These operators create new types from the basic types:

```
*        pointer to
*const   constant pointer to
&        reference to
[]       vector of
()       function returning
```

For example,

```
char* p;
char *const q;
char v[10];
int f(char*);
```

declares p as a pointer to character, q as a constant pointer to character, v as a vector of 10 characters, and f as a function taking an argument of type char* and returning an integer. A pointer variable can hold the address of an object of the appropriate type. For example:

```
char c;
p = &c;
```

Unary & is the "address of" operator.

The name of a vector doubles as the name of its first element, so given the declarations above you could write:

```
p = v;
x = f(v);
x = f(p);
```

The first statement could alternatively have been written:

```
p=&v[0];
```

since all vectors have zero as their lower bound.

For a more exhaustive treatment of these derived types see reference 1 or 2. Functions will be explained in §4. References will be explained in §10.

3 Expressions and Statements

Except for a minor extension to the syntax of the for statement, C++ statements are identical to those provided by C, but note that in C++ local declarations are statements and can be mixed freely with other statements. Except for the addition of the scope resolution operator :: (§14) C++ expressions are identical to those provided by C. If you know C, please skip this section. The discussion of expressions and statements below is very brief. See reference 1 for more details and examples.

Expressions

C++ has a host of operators that will be explained if and where needed. However, it can be noted that the operators

`!`	`(not)`
`~`	`(complement)`
`&`	`(and)`
`^`	`(exclusive or)`
`|`	`(inclusive or)`
`<<`	`(left logical shift)`
`>>`	`(right logical shift)`

apply to integers, and that there is no separate data type for logical operations.

C++ has an assignment operator `=`, rather than an "assignment statement" as in some languages. Assignments can therefore appear in contexts where one might not expect them. For example `x=sqrt(a=3*x)`. This is often useful; for example `a=b=c` means assign `c` to `b` and then to `a`. Another aspect of the assignment operator is that it can be combined with most binary operators into "assignment operators". For example `x[i+3]*=4` means `x[i+3]=x[i+3]*4` except the expression `x[i+3]` is evaluated only once. This gives a pleasing degree of run-time efficiency without having to resort to optimizing compilers. It is also more concise.

Pointers are used extensively in most C++ programs. The unary `*` operator dereferences a pointer. For example, given `char* p;` `*p` is the character pointed to by `p`. An alternative way of expressing this is `p[0]`. In fact, `p[i]` is defined to mean `*(p+i)`. Not only can a vector name be used as a pointer but a pointer can be used as if it were the name of a vector. A vector name is a constant though, whereas a pointer is a variable unless declared otherwise. The increment operator `++` and the decrement operator `--` are often used for pointers.

Expression statements

The most common form of a statement is an expression statement; it consists of an expression followed by a semicolon. For example:

```
a = b*3+c;
cout<<"go go go";
lseek(fd,0,2);
```

Null statements

The simplest statement is the null statement,

```
;
```

it does nothing. It can, however, be useful when the syntax requires a statement, but you have no need for one.

Blocks

A block is a possibly empty list of statements enclosed in curly braces. For example:

```
{ a=b+2; b++; }
```

It enables you to treat several statements as one. The scope of a name declared in a block extends from the point of declaration to the end of the block. It can be "hidden" by declarations of the same name in inner blocks.

If statements

The following example performs both inch to centimeter and centimeter to inch conversion; you are supposed to indicate the unit of the input by appending `i` for inches or `c` for centimeters:

```
#include <stream.h>

main()
{
    const float fac = 2.54;
    float x, in, cm;
    char ch = 0;
    cout<<"enter length: ";
    cin>>x>>ch;
    if (ch == 'i') {
        in = x;
        cm = x*fac;
    }
    else if (ch == 'c') {
        in = x/fac;
        cm = x;
    }
    else
        in = cm = 0;
    cout<<in<<" in = "<<cm<<" cm\n";
}
```

As can be seen the condition in an if-statement must be parenthesized. The `else` part may be omitted. For the input `10i` this program will produce

```
10 in = 25.4 cm
```

Switch statements

A switch-statement tests a value against a set of constants. The tests in the example above could have been written like this:

```
switch (ch) {
case 'i':
    in = x;
    cm = x*fac;
    break;
case 'c':
    in = x/fac;
    cm = x;
    break;
default:
    in = cm = 0;
    break;
}
```

The `break` statements are used to exit the switch-statement. The case constants must be distinct, and if the value tested does not match any of them the `default` is chosen.

While statements

Consider copying a string given a pointer `p` to its first character and a pointer `q` the target. By convention a string is terminated by the character `NULL`.

```
while (*p != NULL) {
    *q = *p;
    q = q+1;
    p = p+1;
}
*q = NULL;
```

The condition following `while` must be parenthesized. The condition is evaluated, and if its value is non-zero the statement directly following is executed. This carries on until the condition evaluates to zero.

This example is rather verbose. The operator ++ can be used to express increment directly, and the test can be simplified using the observation that the value of `NULL` is zero:

```
while (*p) *q++ = *p++;
*q = 0;
```

where the construct `*p++` means: "take the character pointed to by `p` then increment `p`".

The example can be further compressed since the pointer `p` is dereferenced twice each time round the loop. The character copy can be performed at the same time as the condition is tested:

```
while (*q++ = *p++) ;
```

which takes the character pointed to by `p`, increments `p`, copies that character to the location pointed to by `q` and increments `q`. If the character is non-zero, the loop is repeated. Since all the work is done in the condition, no statement is needed. The null-statement is used to indicate this. C is both loved and hated for enabling such extremely terse expression oriented coding.

For statements

Consider copying ten elements from one vector to another:

```
for (int i=0; i<10; i++) q[i]=p[i];
```

This is equivalent to

```
int i = 0;
while (i<10) {
    q[i] = p[i];
    i++;
}
```

but more readable since all the information controlling the loop is localized. The first part of a for-statement need not be a declaration, any statement will do. For example:

```
for (i=0; i<10; i++) q[i]=p[i];
```

is again equivalent provided `i` is suitably declared earlier.

4 Functions

A function is a named part of a program that can be invoked from other parts of the program as often as needed. For example, consider writing out powers of 2:

```
float pow(float,int);

main()
{
    for (int i=0; i<10; i++) cout<<pow(2,i)<<"\n";
}
```

The first line is a function declaration specifying `pow` to be a function taking a `float` and an `int` argument returning a `float`. A function declaration is used wherever the type of a function defined elsewhere is needed.

In a call each function argument is checked against its expected type exactly as if a variable of the declared type were being initialized. This ensures proper type checking and type conversion. For example, a call `pow(12.3,"10")` will cause the compiler to complain because `"10"` is a string and not an `int`, and for the call `pow(2,i)` the compiler will convert the integer constant 2 to a `float` as expected by the function.

`Pow` might be defined as a power function like this:

```
float pow(float x, int n)
{
    if (n < 0) error("sorry, cannot handle negative exponents");

    switch (n) {
    case 0:   return 1;
    case 1:   return x;
    default:  return x*pow(x,n-1);
    }
}
```

The first part of a function definition specifies the name of the function, the type of the value it returns (if any), and the types and names of its arguments (if any). A value is returned from a function using a return-statement as shown.

Different functions typically have different names, but for functions performing similar tasks on different types of objects it is sometimes nicer to let these functions have the same name. When their argument types are different the compiler can distinguish them anyway. For example, one could have one power function for integers and another for floating point variables:

```
overload pow;
int pow(int,int);
double pow(double,double);
...
x = pow(2,10);
y = pow(2.0,10.0);
```

5 Program structure

A C++ program typically consists of many source files, each containing a sequence of declarations of types, functions, variables, and constants. For a name to refer to the same thing in two source files it must be declared to be external. For example:

```
extern double sqrt(double);
extern istream cin;
```

The most common way of guaranteeing consistency between source files is to place such declarations in separate files, called "header files", and then "include", that is copy, those header files in all files needing the declarations. For example, if the declaration of `sqrt` was stored in the header file for the standard mathematical functions `math.h`, and you wanted to take the square root of 4 you could write:

```
#include <math.h>
x = sqrt(4);
```

Since a typical header file is included into many source files it does not contain declarations that should not be replicated. For example, function bodies are only provided for inline functions (§13) and initializers only for constants (§2.1). Except for those cases, a header file is a repository for type information; it provides an interface between separately compiled parts of a program.

In an include directive a file name enclosed in angle brackets like `<math.h>` above refers to the file of that name in a "standard include directory"; files elsewhere are referred to by names enclosed in double quotes. For example

```
#include "math1.h"
#include "/usr/bs/math2.h"
```

would include `math1.h` from the user's current directory and `math2.h` from the directory `/usr/bs`.

6 Structures

Let us define a new type `ostream` to represent an output stream. The first version is trivially simple, but it will be refined until you get a feel for the real `ostream` used in the stream i/o system. The idea is to put characters into a buffer `buf` until it is full and then write `buf` to a file `file`:

```
struct ostream {
    FILE* file;
    int nextchar;
    char buf[128];
};
```

You can now declare an output stream like this:

```
ostream my_out = { stdout, 0 };
```

The construct

```
= { ... }
```

is an initializer. The members of `my_out` are initialized in order, so that `my_out.file` is `stdout`, `my_out.nextchar` is zero and `my_out.buf` uninitialized. (The `.` (dot) operator is used to access a member of a structure; `stdout` is the "standard output stream" of the underlying operating system. The basic output operation `write` can be used for `stdout`).

A simple character output function can be defined for an `ostream` like this:

```
void putchar(ostream* s, char ch)
{
    if (s->nextchar==128) {
        write(fileno(s->file),s->buf,128);
        s->nextchar = 0;
    }
    s->buf[s->nextchar++] = ch;
}
```

The keyword `void` is used to indicate that `putchar` does not return a value. As shown, the `->` operator is used to get to a member of a structure given a pointer. This code is sloppy (why?), but will actually handle the simplest cases; by writing

```
putchar(&my_out,'H');
putchar(&my_out,'e');
```

you could eventually manage to say `Hello, world`.

Naturally you would next define a function like

```
void putstring(ostream* s, char* p)
{
    for (int i = 0; p[i]; i++) putchar(s,p[i]);
}
```

and a `putlong`, and a `putdouble`, etc.

7 Problems

Proceeding as described above you could get a quite acceptable i/o system: C standard i/o is designed along this line. To save writing you would add functions that implicitly applied to the most common output stream. For example:

```
void mputlong(long i)
{
    putlong(&my_stream,i);
}
```

These functions produce a character string representation of their arguments. Versions that write that representation onto a string instead of a file are also useful:

```
void sputlong(char* s, long i)
{
    ...
}
```

However, there are problems. The most obvious, the proliferation of function names, can be handled simply by giving them all the same name:

```
overload put;
void put(ostream*, char*);
void put(ostream*, long);
void put(ostream*, double);
...
void put(char*);
void put(long);
...
void put(char*, char*);
...
```

Worse, there is no formal connection between these put functions and type ostream. Suppose you wanted to change the representation of an ostream. In any but the smallest program there is no easy way of finding all the places a member of ostream was used, and supposing ostream was a type used by many programs, how would you find the programs needing modification after even the most trivial change? Reversing the order of declaration of file and nextchar would potentially affect every program on your system, and would also invalidate every initializer.

8 Classes

A solution is to split the declaration of ostream into two parts: a private part holding information only needed by its implementer, and a public part presenting an interface to the general public:

```
class ostream {
    FILE* file;
    int nextchar;
    char buf[128];
    void putchar(char);
public:
    put(char*);
    put(long);
    put(double);
};
```

Now a user can only call those three put functions, and only those can use the names of the data members. In other words a class is a struct whose members are private unless their declarations appear after the label public. For example

```
my_stream.put("Hello, world\n");
```

calls put using the usual syntax for members. A member function can only be called for a specified object of its type. When in a member function, the object for which the function was called is accessed through a pointer called this. In a member function of class C, the keyword this is implicitly defined as

```
C* this;
```

You can now write

```
void ostream.put(char* p)
{
    while (*p) this->putchar(*p++);
}
```

The ostream. prefix is necessary to distinguish ostream's put from functions called put in other classes. The function body can be simplified, however, since this use of `this` is optional; in a member function, member names used without qualification refer to the object for which the function was called.

```
void ostream.put(char* p)
{
    while (*p) putchar(*p++);
}
```

would have been enough, and that is the more typical way of writing member functions. Consequently, most uses of `this` are implicit.

A `struct` is actually defined as a `class` with all members public, so a `struct` can have member functions too.

Since the representation of an `ostream` now is private, output functions for user-defined types must be written in terms of the basic put functions. For example, if you had a type `complex` you could define a put function for it:

```
void put(ostream* s, complex z)
{
    s->put("(");
    s->put(z.real);
    s->put(",");
    s->put(z.imag);
    s->put(")");
}
```

It could be called like this:

```
complex z
put(&my_stream, z);
```

This is actually not very nice: the syntax for printing a value of a "basic" type is different from the one needed to print a value of a user-defined type. Furthermore, you need to write a separate call for each value.

9 Operator Overloading

Both problems can be overcome by using an output operator rather than an output function. To define a C++ operator @ for a user-defined type you define a function called `operator@` which takes arguments of the appropriate type. For example:

```
class ostream {
    ...
    ostream operator<<(char*);
};

ostream ostream.operator<<(char* p)
{
    while (*p) putchar(*p++);
    return *this;
}
```

defines the `<<` operator as a member of class `ostream`, so that `s<<p` will be interpreted as `s.operator<<(p)` when `s` is an `ostream` and `p` is a character pointer. Returning the `ostream` as the return value enables you to apply `<<` to the result of an output operation. For example `s<<p<<q` is interpreted as `s.operator<<(p).operator<<(q)`. This is the way output operations are provided for the "basic" types. Using the set of operations provided as public members of class `ostream`, you can now define `<<` for a user-defined type like `complex` without modifying the declaration of class `ostream`:

```
ostream operator<<(ostream s, complex z)
{
    return s<<"("<<z.real<<","<<z.imag<<")";
}
```

This will write the values out in the right order since <<, like most C++ operators, groups left-to-right; that is, a<<b<<c means (a<<b)<<c. The compiler knows the difference between member functions and non-member functions when interpreting operators. For example, if z is a complex variable s<<z will be expanded using the standard (non-member) function call operator<<(s,z).

10 References

This last version of ostream unfortunately contains a serious error and is furthermore very inefficient. The problem is that the ostream is copied twice for each use of <<: once as an argument and once as the return value. This leaves nextchar unchanged after a call (the example above does work correctly, however; why?). A facility for passing a pointer to that ostream rather than the ostream itself is needed.

A reference to a type T, written T&, can be initialized by either an object of that type or a pointer to one; in the former case the address of operator & is implicitly applied. For example:

```
ostream& s1 = my_stream;
ostream& s2 = &my_stream;
```

Both s1 and s2 now refer to my_stream and can be used as alternative names for it. When used, a reference refers to the variable with which it has been initialized. For example, assignment

```
s1 = s2;
```

copies the object referred to, in this case my_stream. Members are selected using the dot operator

```
s1.put("don't use ->");
```

and if you apply the address operator you get the address of the object referred to:

```
&s1 == &my_stream
```

The first obvious use of references is to ensure that a pointer rather than the object itself is passed to an output function (this is called "call by reference" in some other languages):

```
ostream& operator(ostream& s, complex z) {
    return s<<"("<<z.real<<","<<z.imag<<")";
}
```

Interestingly enough the body of this function is unchanged, but had you assigned to s you would now have affected the object given as the argument itself rather than a copy. In this case, returning a reference also improves efficiency.

References are also essential for the definition of input streams, since the input operator is given the variable to read into as an operand.

```
class istream {
    ...
    int state;
public:
    istream& operator>>(char&);
    istream& operator>>(char*);
    istream& operator>>(int&);
    istream& operator>>(long&);
    ...
};
```

Note that istream needs more functions than ostream, since type conversion applies to basic types like int and long, but not to pointers to those types.

11 Constructors

The definition of `ostream` as a class made the data members private. In particular it rendered the initialization

```
ostream my_stream = { stdout, 0 };
```

illegal. Only a member function can access the private members, so you must provide one for initialization. Such a function is called a constructor and is distinguished by having the same name as its class:

```
class ostream {
    ...
    ostream(FILE* fp);
    ostream(int size, char* s);
};
```

Here two were provided; one takes a file descriptor like `stdout` above for real output; the other takes a character pointer and a size for string formatting.

You can now declare streams like this:

```
ostream my_stream(stdout);
char xx[256];
ostream xx_stream(256,xx);
```

Providing a class with a constructor not only provides a way of initializing objects, but also ensures that all objects of that class will be initialized. It is not possible to declare a variable of a class with a constructor without a constructor being called. If a class has a constructor that does not take arguments, that constructor will be called if no arguments are given in the declaration.

12 Vectors

The vector concept built into C++ was designed (for C) to allow maximal run-time efficiency and minimal store overhead. It is also, especially when used together with pointers, an extremely versatile tool for building "higher level" facilities. You could, however, complain that a vector size must be specified as a constant, that there is no vector bounds checking, etc. The answer to such complaints is "you can program that yourself". Let us therefore test C++'s abstraction facilities by trying to provide these features for vector types of our own design and observe the difficulties involved, the costs incurred, and the convenience of use of the resulting vector types.

```
class vector {
    int*v;
    int sz;
public:
    vector(int);
    ~vector();
    int size() { return sz; }
    void set_size(int);
    int& operator[](int);
    int& elem(int i) { return &v[i]; }
};
```

The function `size` returns the number of elements of the vector, that is indices must be in the range `[0..size()-1]`; `set_size` is provided to change that size; `elem` provides access to members without checking the index, and `operator[]` provides access with bounds check.

The idea is to have the class itself be a fixed sized structure controlling access to the actual vector storage which is allocated by the vector constructor using the free store allocator operator `new`:

```
vector.vector(int s)
{
    if (s<=0) error("bad vector size");
    sz = s;
    v = new int[s];
}
```

You can now declare vectors very nearly as elegantly as "real vectors":

```
vector v1(100);
vector v2(nelem*2-4);
```

The access operation can be defined as

```
int& vector.operator[](int i) {
    if (0<=i && i<sz) return &v[i];
    error("vector index out of range")
}
```

The operator && (andand) is a logical-and operator. Its right hand operand is only evaluated if necessary, that is, provided its left hand operand does not evaluate to zero. Returning a reference ensures that the [ ] notation can be used on either side of an assignment:

```
v1[x] = v2[y];
```

The function with the funny name ~vector is a destructor. A destructor is called implicitly when a class object goes out of scope, so if you define it like this:

```
vector.~vector()
{
    delete v;
}
```

it will, using the delete operator, deallocate the space allocated by the constructor, so that when a vector goes out of scope all its space is reclaimed for potential re-use.

13 Inline expansion

Given the frequency of very small functions you might worry about the cost of function calls. A member function is no more expensive to call than a non-member function with the same number of arguments (remembering that a member function always has at least one argument), and C++ function calls are about as efficient as you can get for any language. However, for extremely small functions the call overhead can become an issue. If so, you might consider specifying a function to be "inline expanded". If you do, the compiler will try to generate the proper code for the function at the place of the call. The semantics of the call is unchanged. For example:

```
vector s(100);
...
i = s.size()
x = elem(i-1);
```

will generate code equivalent to

```
i = 100;
x = s.v[i-1];
```

The C++ compiler is usually smart enough to generate code as good as you would have got from straightforward macro expansion. Naturally it will sometimes have to use temporary variables and other little tricks to preserve the semantics.

You can indicate that you want a function inline expanded by preceding its definition by the keyword inline, or, for a member function, simply by including the function definition in the class declaration, as was done for size() and elem() above.

Inline functions slow down compilation and clutter class declarations so they should be avoided when they are not necessary. For inline substitution to be a significant benefit for a function the function must be very small. When used well `inline` functions simultaneously increase the running speed and decrease the object code size.

14 Derived classes

Now let us define a vector for which a user can define the index bounds:

```
class vec: public vector {
    int low, high;
public:
    vec(int, int);
    int& elem(int);
    int& operator[](int);
}
```

Defining `vec` as

```
: public vector
```

means that first of all a `vec` is a `vector`. That is, type `vec` has all the properties of type `vector` in addition to the ones declared specifically for it. Class `vector` is said to be the "base" class for `vec`, and conversely `vec` is said to be "derived" from `vector`.

Class `vec` modifies class `vector` by providing a different constructor, requiring the user to specify the two index bounds rather than the size, and by providing its own access functions `elem()` and `operator[]()`. A `vec`'s `elem()` is easily expressed in terms of `vector`'s `elem()`:

```
int& vec.elem(int i)
{
    return vector::elem(i-low);
}
```

The scope resolution operator `::` is used to avoid getting caught in an infinite recursion by calling `vec::elem()` from itself (unary `::` can be used to refer to global names).

The constructor can be written like this:

```
vec.vec(int lb, int hb) : (hb-lb+1)
{
    if (hb-lb<0) hb = lb;
    low = lb;
    high = hb;
}
```

The construct `:(hb-lb+1)` is used to specify the argument list needed for the base class constructor `vector()`.

This line of development of the vector type can be explored further. It is quite simple to provide multi-dimensional arrays (overload `()` as the access function), arrays where the number of dimensions is specified as an argument to a constructor, Fortran style arrays that can be accessed both as having two and three dimensions etc.

Such a class controls access to some data. Since all access is through the interface provided by the public part of the class, the representation of the data can in fact be changed arbitrarily to suit the needs of the implementer. For example, it would be trivial to change the representation of a vector to a linked list. The other side of this coin is that any suitable interface for a given implementation can be provided.

15 More about operators

An alternative direction of development is to provide vectors with operations:

```
struct Vec : public vector {
    Vec(int);
    Vec(Vec&);
    ~Vec();
    void operator=(Vec&);
    void operator*=(Vec&);
    void operator*=(int);
    ...
};
```

Since a Vec has no private members (except the ones inherited from vector) it can be specified as a struct. The assignment operator is overloaded, and can be defined like this:

```
void Vec.operator=(Vec& a)
{
    int s = size();
    if (s!=a.size()) error("bad vector size for =");
    for (int i = 0; i<s; i++) v[i]=a.v[i];
}
```

Assignment of Vecs now truly copies the elements, whereas assignment of vectors simply copies the structure controlling access to the elements. However, the latter also happens when a vector is copied without explicit use of the assignment operator: (1) when a vector is initialized by assignment of another vector, (2) when a vector is passed as an argument, and (3) when a vector is passed as the return value from a function. To gain control in these cases for Vec vectors you define the constructor:

```
Vec.Vec(Vec& a) : (a.size())
{
    for (int i = 0; i<sz; i++) v[i]=a.v[i];
}
```

This constructor initializes a vector as the copy of another, and will be called in the cases mentioned before.

For operators like = and += the expression on the left is clearly "special" and it seems natural to implement them as operations on the object denoted by that expression. In particular, it is then possible to change that object's value. For operators like + and - the left hand operand does not appear to need special attention. You could, for example, pass both arguments by value and still get a correct implementation of +:

```
Vec operator+(Vec a, Vec b)
{
    int s = a.size();
    if (s != b.size()) error("bad vector size for +");
    Vec& sum = new Vec(s);
    for (int i = 0; i<s; i++) sum.elem(i) = a.elem(i) + b.elem(i);
    return sum;
}
```

This function does not operate directly on the representation of a vector, indeed it couldn't since it is not a member. However, it is sometimes desirable to allow non-member functions to access the private part of a class object. For example, had there been no "unchecked access" function, vector::elem(), you would have been forced to check the index i against the vector bounds three times every time round the loop. This problem was avoided here, but it is typical, so there is a mechanism for a class to grant access to its private part to a non-member function. A declaration of the function preceded by the keyword friend is simply placed in the declaration of the class. For example, given

```
class vector {
    ...
    friend Vec operator+(Vec, Vec);
};
```

you could have written:

```
Vec operator+(Vec a, Vec b)
{
    int s = a.size();
    if (s != b.size()) error("bad vector size for +");
    Vec& sum = new Vec(s);
    int* sp = sum.v;
    int* ap = a.v;
    int* bp = b.v;
    while (s--) *sp++ = *ap++ + *bp++;
    return sum;
}
```

One particularly useful aspect of the `friend` mechanism is that a function can be the friend of two or more classes. To see this consider defining a `vector` and `matrix` and then defining a multiplication function.

16 Generic vectors

"So far so good", you might say, "but I want one of those vectors for the type `matrix` I just defined". Unfortunately, C++ does not provide a facility for defining a class vector with the type of the elements as an argument. One way to proceed is to replicate the definition of both the class and its member functions. This is not ideal, but often acceptable.

You can use a macro processor to mechanize that task. For example, class `vector` presented above is a simplified version of a class that can be found in a standard header file. You could write:

```
#include <vector.h>

declare(vector,int);

main()
{
    vector(int) vv(10);
    vv[2] = 3;
    vv[10] = 4;    /* range error */
}

implement(vector,int);
```

The file `vector.h` defines macros so that `declare(vector,int)` expands to the declaration of a class `vector` very much like the one defined above, and `implement(vector,int)` expands to the definitions of the functions of that class. Since `implement(vector,int)` expands into function definitions it can only be used once in a program, whereas `declare(vector,int)` must be used once in every file manipulating this kind of integer vectors.

```
declare(vector,char);
implement(vector,char);
```

would give you a (separate) type of vector of characters.

17 Polymorphic vectors

Alternatively you might define your vector and other "container classes" in terms of pointers to objects:

```
class cvector {
    common** v;
    ...
public:
    common*& elem(int);
    common*& operator[](int);
    ...
};
```

Note that since pointers and not the objects themselves are stored in such vectors an object can be "in" several such vectors at the same time. This is a very useful feature for container classes like vectors, linked lists, classes, etc.

Furthermore, a pointer to a derived class can be assigned to a pointer to its base class, so the `cvector` above can be used to hold pointers to objects of all classes derived from `common`. For example:

```
class apple : public common { ... };
class orange : public common { ... };
cvector fruitbowl(100);
apple aa;
orange oo;
fruitbowl[0] = &aa;
fruitbowl[1] = &oo;
```

However, the exact type of an object entered into such a container class is no longer known by the compiler. For example, in the example above you know that an element of the vector is a `common`, but is it an `apple` or an `orange`? Typically that exact type must be recovered later in order to use the object correctly. To do this you must either store some form of type information in the object itself or ensure that only objects of a given type are put in the container. The latter is trivially achieved using a derived class. For example, you could make a vector of `apple` pointers:

```
class apple_vector : public cvector {
public:
    apple*& elem(int i) { return (apple*&) cvector::elem(i); }
    ...
};
```

using the "type casting" notation `(type)expression` to convert the `common*&` (a reference to a pointer to a `common`) returned by `cvector::elem` to an `apple*&`. The alternative, storing type identification in each object, brings us to the style of programming referred to as "object based".

18 Virtual functions

Consider writing a program for displaying shapes on a screen. The common attributes of shapes will be represented by class `shape`, specific attributes by specific derived classes:

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

Functions that can be defined without knowledge of the specific shape (for example `move`, and `where`), can be declared as usual. The rest is declared `virtual`, that is to be defined in a derived class. For example:

```
class circle: public shape {
    int radius;
public:
    void draw();
    void rotate(int i) {}
    ...
};
```

Now if `shape_vec` is a vector of shapes as defined above you can write:

```
for (int i = 0; i<no_of_shapes; i++) shape_vec[i].rotate(45);
```

to rotate (and re-draw) all shapes 45 degrees.

This style is extremely useful in all interactive programs where "objects" of various types are treated in a uniform manner by the basic software. In a sense the typical operation is for the user to point to some object and say *Who are you? What are you?* or *Do your stuff!* without providing type information. The program can and must figure that out for itself.

19 Compatibility

C++ is not completely compatible with C, but it comes very close. In C++

```
int f();
```

declares a function that does not accept arguments; in C it declares a function that takes any number of arguments of any types (in C++, that can be stated as `int f(...)`). In C++ the scope a non-local name is limited to its source file; in C every non-local name is "external". Consequently, to compile a C program as a C++ program you typically (only) need re-write your own header files (there are already C++ versions of the standard ones). In addition, C++ has 11 more keywords than C; these cannot be used as names of variables, etc. You can link C and C++ object files together without modification.

20 Efficiency

Run time efficiency of the generated code and compactness of the representation of user defined data structures was considered of primary importance in the design of C++. A call of a member function is as fast as an equivalent (C++ or C) non-member function call with the same number of arguments. A call of a virtual function typically involves only three memory references extra. The representation of a class object takes up only the space needed for the data members specified by the user (allocated conforming to machine dependent alignment requirements). When virtual functions are declared for a class, objects of that class will containing one extra hidden pointer.

21 Caveat

Experienced C programmers have ended up with perfectly awful C++ programs because they immediately started using all the new features at once. It is worth remembering that most programs are best written without operator overloading, and using only a few examples of inline functions, private data, friends, references, derived classes, and virtual functions. Proceed with caution.

22 Acknowledgements

C++, as presented here could never have matured without the constant help and constructive criticism of my colleagues and users; notably Tom Cargill, Jim Coplien, Stu Feldman, Sandy Fraser, Steve Johnson, Brian Kernighan, Bart Locanthi, Doug McIlroy, Dennis Ritchie, Ravi Sethi, and Jon Shopiro. .C++ clearly owes most to C; the influence of Simula67 is pervasive in the class facilities, and you may also notice Algol68 like facilities.

23 References

[1] B.Kernighan and D.M.Ritchie: The C programming Language, Prentice-Hall 1978.
[2] B.Stroustrup: The C++ Programming Language - Reference Manual, In this volume.
[3] B.Stroustrup: Data Abstraction in C, In this volume.