2

A Tour of C++

The first thing we do, let´s
kill all the language lawyers.
– Henry VI, part II

What is C++? — programming paradigms — procedural programming — modularity — separate compilation — exception handling — data abstraction — user-defined types — concrete types — abstract types — virtual functions — object-oriented programming — generic programming — containers — algorithms — language and programming — advice.

2.1 What is C++?

C++ is a general-purpose programming language with a bias towards systems programming that

- is a better C,
- supports data abstraction,
- supports object-oriented programming, and
- supports generic programming.

This chapter explains what this means without going into the finer details of the language definition. Its purpose is to give you a general overview of C++ and the key techniques for using it, *not* to provide you with the detailed information necessary to start programming in C++.

If you find some parts of this chapter rough going, just ignore those parts and plow on. All will be explained in detail in later chapters. However, if you do skip part of this chapter, do yourself a favor by returning to it later.

Detailed understanding of language features – even of *all* features of a language – cannot compensate for lack of an overall view of the language and the fundamental techniques for using it.

2.2 Programming Paradigms

Object-oriented programming is a technique for programming – a paradigm for writing ‘‘good’’ programs for a set of problems. If the term ‘‘object-oriented programming language’’ means anything, it must mean a programming language that provides mechanisms that support the object-oriented style of programming well.

There is an important distinction here. A language is said to *support* a style of programming if it provides facilities that make it convenient (reasonably easy, safe, and efficient) to use that style. A language does not support a technique if it takes exceptional effort or skill to write such programs; it merely *enables* the technique to be used. For example, you can write structured programs in Fortran77 and object-oriented programs in C, but it is unnecessarily hard to do so because these languages do not directly support those techniques.

Support for a paradigm comes not only in the obvious form of language facilities that allow direct use of the paradigm, but also in the more subtle form of compile-time and/or run-time checks against unintentional deviation from the paradigm. Type checking is the most obvious example of this; ambiguity detection and run-time checks are also used to extend linguistic support for paradigms. Extra-linguistic facilities such as libraries and programming environments can provide further support for paradigms.

One language is not necessarily better than another because it possesses a feature the other does not. There are many examples to the contrary. The important issue is not so much what features a language possesses, but that the features it does possess are sufficient to support the desired programming styles in the desired application areas:

[1] All features must be cleanly and elegantly integrated into the language.

[2] It must be possible to use features in combination to achieve solutions that would otherwise require extra, separate features.

[3] There should be as few spurious and ‘‘special-purpose’’ features as possible.

[4] A feature’s implementation should not impose significant overheads on programs that do not require it.

[5] A user should need to know only about the subset of the language explicitly used to write a program.

The first principle is an appeal to aesthetics and logic. The next two are expressions of the ideal of minimalism. The last two can be summarized as ‘‘what you don’t know won’t hurt you.’’

C++ was designed to support data abstraction, object-oriented programming, and generic programming in addition to traditional C programming techniques under these constraints. It was *not* meant to force one particular programming style upon all users.

The following sections consider some programming styles and the key language mechanisms supporting them. The presentation progresses through a series of techniques starting with procedural programming and leading up to the use of class hierarchies in object-oriented programming and generic programming using templates. Each paradigm builds on its predecessors, each adds something new to the C++ programmer’s toolbox, and each reflects a proven design approach.

The presentation of language features is not exhaustive. The emphasis is on design approaches and ways of organizing programs rather than on language details. At this stage, it is far more important to gain an idea of what can be done using C++ than to understand exactly how it can be achieved.

2.3 Procedural Programming

The original programming paradigm is:

> *Decide which procedures you want;*
> *use the best algorithms you can find.*

The focus is on the processing – the algorithm needed to perform the desired computation. Languages support this paradigm by providing facilities for passing arguments to functions and returning values from functions. The literature related to this way of thinking is filled with discussion of ways to pass arguments, ways to distinguish different kinds of arguments, different kinds of functions (e.g., procedures, routines, and macros), etc.

A typical example of ''good style'' is a square-root function. Given a double-precision floating-point argument, it produces a result. To do this, it performs a well-understood mathematical computation:

```
double sqrt(double arg)
{
    // code for calculating a square root
}

void f()
{
    double root2 = sqrt(2);
    // ...
}
```

Curly braces, `{ }`, express grouping in C++. Here, they indicate the start and end of the function bodies. The double slash, `//`, begins a comment that extends to the end of the line. The keyword ***void*** indicates that a function does not return a value.

From the point of view of program organization, functions are used to create order in a maze of algorithms. The algorithms themselves are written using function calls and other language facilities. The following subsections present a thumb-nail sketch of C++'s most basic facilities for expressing computation.

2.3.1 Variables and Arithmetic

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 specifies a type for that name. A *type* defines the proper use of a name or an expression.

C++ offers a variety of fundamental types, which correspond directly to hardware facilities. For example:

```
bool    // Boolean, possible values are true and false
char    // character, for example, 'a', 'z', and '9'
int     // integer, for example, 1, 42, and 1216
double  // double-precision floating-point number, for example, 3.14 and 299793.0
```

A ***char*** variable is of the natural size to hold a character on a given machine (typically a byte), and an ***int*** variable is of the natural size for integer arithmetic on a given machine (typically a word).

The arithmetic operators can be used for any combination of these types:

```
+   // plus, both unary and binary
-   // minus, both unary and binary
*   // multiply
/   // divide
%   // remainder
```

So can the comparison operators:

```
==  // equal
!=  // not equal
<   // less than
>   // greater than
<=  // less than or equal
>=  // greater than or equal
```

In assignments and in arithmetic operations, C++ performs all meaningful conversions between the basic types so that they can be mixed freely:

```
void some_function()    // function that doesn't return a value
{
    double d = 2.2;     // initialize floating-point number
    int i = 7;          // initialize integer
    d = d+i;            // assign sum to d
    i = d*i;            // assign product to i
}
```

As in C, = is the assignment operator and == tests equality.

2.3.2 Tests and Loops

C++ provides a conventional set of statements for expressing selection and looping. For example, here is a simple function that prompts the user and returns a Boolean indicating the response:

```
bool accept()
{
    cout << "Do you want to proceed (y or n)?\n";    // write question

    char answer = 0;
    cin >> answer;                                   // read answer

    if (answer == 'y') return true;
    return false;
}
```

The << operator (''put to'') is used as an output operator; ***cout*** is the standard output stream. The >> operator (''get from'') is used as an input operator; ***cin*** is the standard input stream. The type of the right-hand operand of >> determines what input is accepted and is the target of the input operation. The ***\n*** character at the end of the output string represents a newline.

The example could be slightly improved by taking an 'n' answer into account:

```
bool accept2()
{
    cout << "Do you want to proceed (y or n)?\n";    // write question

    char answer = 0;
    cin >> answer;                                   // read answer

    switch (answer) {
    case 'y':
        return true;
    case 'n':
        return false;
    default:
        cout << "I'll take that for a no.\n";
        return false;
    }
}
```

A *switch-statement* tests a value against a set of constants. The case constants must be distinct, and if the value tested does not match any of them, the ***default*** is chosen. The programmer need not provide a ***default***.

Few programs are written without loops. In this case, we might like to give the user a few tries:

```
bool accept3()
{
    int tries = 1;
    while (tries < 4) {
        cout << "Do you want to proceed (y or n)?\n";    // write question
        char answer = 0;
        cin >> answer;                                   // read answer

        switch (answer) {
        case 'y':
            return true;
        case 'n':
            return false;
        default:
            cout << "Sorry, I don't understand that.\n";
            tries = tries + 1;
        }
    }
    cout << "I'll take that for a no.\n";
    return false;
}
```

The *while-statement* executes until its condition becomes ***false***.

2.3.3 Pointers and Arrays

An array can be declared like this:

```
char v[10];     // array of 10 characters
```

Similarly, a pointer can be declared like this:

```
char* p;    // pointer to character
```

In declarations, `[]` means ''array of'' and `*` means ''pointer to.'' All arrays have `0` as their lower bound, so `v` has ten elements, `v[0]`...`v[9]`. A pointer variable can hold the address of an object of the appropriate type:

```
p = &v[3];      // p points to v's fourth element
```

Unary `&` is the address-of operator.

Consider copying ten elements from one array to another:

```
void another_function()
{
    int v1[10];
    int v2[10];
    // ...
    for (int i=0; i<10; ++i) v1[i]=v2[i];
}
```

This *for-statement* can be read as ''set `i` to zero, while `i` is less than `10`, copy the `i`th element and increment `i`.'' When applied to an integer variable, the increment operator `++` simply adds `1`.

2.4 Modular Programming

Over the years, the emphasis in the design of programs has shifted from the design of procedures and toward the organization of data. Among other things, this reflects an increase in program size. A set of related procedures with the data they manipulate is often called a *module*. The programming paradigm becomes:

> *Decide which modules you want;*
> *partition the program so that data is hidden within modules.*

This paradigm is also known as the *data-hiding principle*. Where there is no grouping of procedures with related data, the procedural programming style suffices. Also, the techniques for designing ''good procedures'' are now applied for each procedure in a module. The most common example of a module is the definition of a stack. The main problems that have to be solved are:

[1] Provide a user interface for the stack (e.g., functions `push()` and `pop()`).

[2] Ensure that the representation of the stack (e.g., an array of elements) can be accessed only through this user interface.

[3] Ensure that the stack is initialized before its first use.

C++ provides a mechanism for grouping related data, functions, etc., into separate namespaces. For example, the user interface of a ***Stack*** module could be declared and used like this:

```
namespace Stack {          // interface
    void push(char);
    char pop();
}

void f()
{
    Stack::push('c');
    if (Stack::pop() != 'c') error("impossible");
}
```

The ***Stack***:: qualification indicates that the ***push***() and ***pop***() are those from the ***Stack*** namespace. Other uses of those names will not interfere or cause confusion.

The definition of the ***Stack*** could be provided in a separately-compiled part of the program:

```
namespace Stack {          // implementation
    const int max_size = 200;
    char v[max_size];
    int top = 0;

    void push(char c) { /* check for overflow and push c */ }
    char pop() { /* check for underflow and pop */ }
}
```

The key point about this ***Stack*** module is that the user code is insulated from the data representation of ***Stack*** by the code implementing ***Stack***::***push***() and ***Stack***::***pop***(). The user doesn't need to know that the ***Stack*** is implemented using an array, and the implementation can be changed without affecting user code. The /* starts a comment that extends to the following */.

Because data is only one of the things one might want to ''hide,'' the notion of data hiding is trivially extended to the notion of *information hiding*; that is, the names of functions, types, etc., can also be made local to a module. Consequently, C++ allows any declaration to be placed in a namespace (§8.2).

This ***Stack*** module is one way of representing a stack. The following sections use a variety of stacks to illustrate different programming styles.

2.4.1 Separate Compilation

C++ supports C's notion of separate compilation. This can be used to organize a program into a set of semi-independent fragments.

Typically, we place the declarations that specify the interface to a module in a file with a name indicating its intended use. Thus,

```
namespace Stack {          // interface
    void push(char);
    char pop();
}
```

would be placed in a file ***stack.h***, and users will *include* that file, called a *header file*, like this:

```
#include "stack.h"              // get the interface

void f()
{
        Stack::push('c');
        if (Stack::pop() != 'c') error("impossible");
}
```

To help the compiler ensure consistency, the file providing the implementation of the ***Stack*** module will also include the interface:

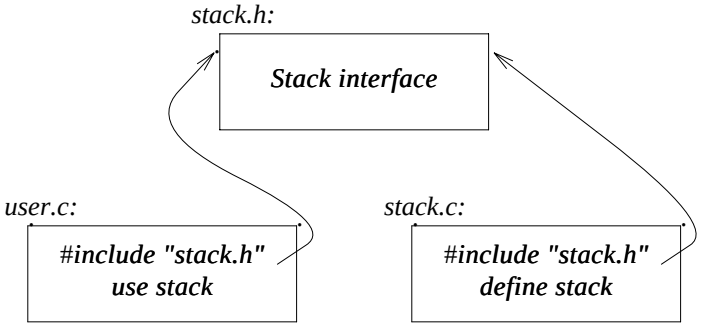
```
#include "stack.h"              // get the interface

namespace Stack {               // representation
        const int max_size = 200;
        char v[max_size];
        int top = 0;
}

void Stack::push(char c) { /* check for overflow and push c */ }

char Stack::pop() { /* check for underflow and pop */ }
```

The user code goes in a third file, say ***user.c***. The code in ***user.c*** and ***stack.c*** shares the stack interface information presented in ***stack.h***, but the two files are otherwise independent and can be separately compiled. Graphically, the program fragments can be represented like this:



Separate compilation is an issue in all real programs. It is not simply a concern in programs that present facilities, such as a ***Stack***, as modules. Strictly speaking, using separate compilation isn't a language issue; it is an issue of how best to take advantage of a particular language implementation. However, it is of great practical importance. The best approach is to maximize modularity, represent that modularity logically through language features, and then exploit the modularity physically through files for effective separate compilation (Chapter 8, Chapter 9).

2.4.2 Exception Handling

When a program is designed as a set of modules, error handling must be considered in light of these modules. Which module is responsible for handling what errors? Often, the module that detects an error doesn't know what action to take. The recovery action depends on the module that invoked

the operation rather than on the module that found the error while trying to perform the operation. As programs grow, and especially when libraries are used extensively, standards for handling errors (or, more generally, ''exceptional circumstances'') become important.

Consider again the ***Stack*** example. What *ought* to be done when we try to ***push***( ) one too many characters? The writer of the ***Stack*** module doesn't know what the user would like to be done in this case, and the user cannot consistently detect the problem (if the user could, the overflow wouldn't happen in the first place). The solution is for the ***Stack*** implementer to detect the overflow and then tell the (unknown) user. The user can then take appropriate action. For example:

```
namespace Stack {          // interface
    void push(char);
    char pop();

    class Overflow { };  // type representing overflow exceptions
}
```

When detecting an overflow, ***Stack***::***push***( ) can invoke the exception-handling code; that is, ''throw an ***Overflow*** exception:''

```
void Stack::push(char c)
{
    if (top == max_size) throw Overflow();
    // push c
}
```

The ***throw*** transfers control to a handler for exceptions of type ***Stack***::***Overflow*** in some function that directly or indirectly called ***Stack***::***push***( ). To do that, the implementation will unwind the function call stack as needed to get back to the context of that caller. Thus, the ***throw*** acts as a multilevel ***return***. For example:

```
void f()
{
    // ...
    try { // exceptions here are handled by the handler defined below
        while (true) Stack::push('c');
    }
    catch (Stack::Overflow) {
        // oops: stack overflow; take appropriate action
    }
    // ...
}
```

The ***while*** loop will try to loop forever. Therefore, the ***catch***-clause providing a handler for ***Stack***::***Overflow*** will be entered after some call of ***Stack***::***push***( ) causes a ***throw***.

Use of the exception-handling mechanisms can make error handling more regular and readable. See §8.3, Chapter 14, Appendix E for further discussion, details, and examples.

2.5 Data Abstraction

Modularity is a fundamental aspect of all successful large programs. It remains a focus of all design discussions throughout this book. However, modules in the form described previously are not sufficient to express complex systems cleanly. Here, I first present a way of using modules to provide a form of user-defined types and then show how to overcome some problems with that approach by defining user-defined types directly.

2.5.1 Modules Defining Types

Programming with modules leads to the centralization of all data of a type under the control of a type manager module. For example, if we wanted many stacks – rather than the single one provided by the ***Stack*** module above – we could define a stack manager with an interface like this:

```
namespace Stack {
    struct Rep;                         // definition of stack layout is elsewhere
    typedef Rep& stack;

    stack create();                     // make a new stack
    void destroy(stack s);              // delete s

    void push(stack s, char c);         // push c onto s
    char pop(stack s);                  // pop s
}
```

The declaration

```
struct Rep;
```

says that ***Rep*** is the name of a type, but it leaves the type to be defined later (§5.7). The declaration

```
typedef Rep& stack;
```

gives the name ***stack*** to a ''reference to ***Rep***'' (details in §5.5). The idea is that a stack is identified by its ***Stack*****::*****stack*** and that further details are hidden from users.

A ***Stack*****::*****stack*** acts much like a variable of a built-in type:

```
struct Bad_pop { };

void f()
{
    Stack::stack s1 = Stack::create();    // make a new stack
    Stack::stack s2 = Stack::create();    // make another new stack

    Stack::push(s1,´c´);
    Stack::push(s2,´k´);

    if (Stack::pop(s1) != ´c´) throw Bad_pop();
    if (Stack::pop(s2) != ´k´) throw Bad_pop();

    Stack::destroy(s1);
    Stack::destroy(s2);
}
```

We could implement this ***Stack*** in several ways. It is important that a user doesn't need to know how we do it. As long as we keep the interface unchanged, a user will not be affected if we decide to re-implement ***Stack***.

An implementation might preallocate a few stack representations and let ***Stack*****::*****create*****()** hand out a reference to an unused one. ***Stack*****::*****destroy*****()** could then mark a representation ''unused'' so that ***Stack*****::*****create*****()** can recycle it:

```
namespace Stack {          // representation

    const int max_size = 200;

    struct Rep {
        char v[max_size];
        int top;
    };

    const int max = 16;  // maximum number of stacks

    Rep stacks[max];     // preallocated stack representations
    bool used[max];      // used[i] is true if stacks[i] is in use

    typedef Rep& stack;
}

void Stack::push(stack s, char c) { /* check s for overflow and push c */ }

char Stack::pop(stack s) { /* check s for underflow and pop */ }

Stack::stack Stack::create()
{
    // pick an unused Rep, mark it used, initialize it, and return a reference to it
}

void Stack::destroy(stack s) { /* mark s unused */ }
```

What we have done is to wrap a set of interface functions around the representation type. How the resulting ''stack type'' behaves depends partly on how we defined these interface functions, partly on how we presented the representation type to the users of ***Stack***s, and partly on the design of the representation type itself.

This is often less than ideal. A significant problem is that the presentation of such ''fake types'' to the users can vary greatly depending on the details of the representation type – and users ought to be insulated from knowledge of the representation type. For example, had we chosen to use a more elaborate data structure to identify a stack, the rules for assignment and initialization of ***Stack*****::*****stack***s would have changed dramatically. This may indeed be desirable at times. However, it shows that we have simply moved the problem of providing convenient stacks from the ***Stack*** module to the ***Stack*****::*****stack*** representation type.

More fundamentally, user-defined types implemented through a module providing access to an implementation type don't behave like built-in types and receive less and different support than do built-in types. For example, the time that a ***Stack*****::*****Rep*** can be used is controlled through ***Stack*****::*****create*****()** and ***Stack*****::*****destroy*****()** rather than by the usual language rules.

2.5.2 User-Defined Types

C++ attacks this problem by allowing a user to directly define types that behave in (nearly) the same way as built-in types. Such a type is often called an *abstract data type*. I prefer the term *user-defined type*. A more reasonable definition of *abstract data type* would require a mathematical ''abstract'' specification. Given such a specification, what are called *types* here would be concrete examples of such truly abstract entities. The programming paradigm becomes:

> *Decide which types you want;*
> *provide a full set of operations for each type.*

Where there is no need for more than one object of a type, the data-hiding programming style using modules suffices.

Arithmetic types such as rational and complex numbers are common examples of user-defined types. Consider:

```
class complex {
    double re, im;
public:
    complex(double r, double i) { re=r; im=i; }   // construct complex from two scalars
    complex(double r) { re=r; im=0; }             // construct complex from one scalar
    complex() { re = im = 0; }                    // default complex: (0,0)

    friend complex operator+(complex, complex);
    friend complex operator-(complex, complex);       // binary
    friend complex operator-(complex);                // unary
    friend complex operator*(complex, complex);
    friend complex operator/(complex, complex);

    friend bool operator==(complex, complex);         // equal
    friend bool operator!=(complex, complex);         // not equal
    // ...
};
```

The declaration of class (that is, user-defined type) ***complex*** specifies the representation of a complex number and the set of operations on a complex number. The representation is *private*; that is, ***re*** and ***im*** are accessible only to the functions specified in the declaration of class ***complex***. Such functions can be defined like this:

```
complex operator+(complex a1, complex a2)
{
    return complex(a1.re+a2.re,a1.im+a2.im);
}
```

A member function with the same name as its class is called a *constructor*. A constructor defines a way to initialize an object of its class. Class ***complex*** provides three constructors. One makes a ***complex*** from a ***double***, another takes a pair of ***double***s, and the third makes a ***complex*** with a default value.

Class ***complex*** can be used like this:

```
void f(complex z)
{
    complex a = 2.3;
    complex b = 1/a;
    complex c = a+b*complex(1,2.3);
    // ...
    if (c != b) c = -(b/a)+2*b;
}
```

The compiler converts operators involving ***complex*** numbers into appropriate function calls. For example, ***c!=b*** means ***operator!=(c,b)*** and ***1/a*** means ***operator/(complex(1),a)***.

Most, but not all, modules are better expressed as user-defined types.

2.5.3 Concrete Types

User-defined types can be designed to meet a wide variety of needs. Consider a user-defined ***Stack*** type along the lines of the ***complex*** type. To make the example a bit more realistic, this ***Stack*** type is defined to take its number of elements as an argument:

```
class Stack {
    char* v;
    int top;
    int max_size;
public:
    class Underflow { };        // used as exception
    class Overflow { };         // used as exception
    class Bad_size { };         // used as exception

    Stack(int s);               // constructor
    ~Stack();                   // destructor

    void push(char c);
    char pop();
};
```

The constructor ***Stack(int)*** will be called whenever an object of the class is created. This takes care of initialization. If any cleanup is needed when an object of the class goes out of scope, a complement to the constructor – called the *destructor* – can be declared:

```
Stack::Stack(int s)         // constructor
{
    top = 0;
    if (s<0 || 10000<s) throw Bad_size();     // "||" means "or"
    max_size = s;
    v = new char[s];        // allocate elements on the free store (heap, dynamic store)
}

Stack::~Stack()             // destructor
{
    delete[] v;             // free the elements for possible reuse of their space (§6.2.6)
}
```

The constructor initializes a new ***Stack*** variable. To do so, it allocates some memory on the free store (also called the *heap* or *dynamic store*) using the ***new*** operator. The destructor cleans up by freeing that memory. This is all done without intervention by users of ***Stack***s. The users simply create and use ***Stack***s much as they would variables of built-in types. For example:

```
Stack s_var1(10);                    // global stack with 10 elements

void f(Stack& s_ref, int i)          // reference to Stack
{
    Stack s_var2(i);                 // local stack with i elements
    Stack* s_ptr = new Stack(20);    // pointer to Stack allocated on free store

    s_var1.push('a');
    s_var2.push('b');
    s_ref.push('c');
    s_ptr->push('d');
    // ...
}
```

This ***Stack*** type obeys the same rules for naming, scope, allocation, lifetime, copying, etc., as does a built-in type such as ***int*** and ***char***.

Naturally, the ***push()*** and ***pop()*** member functions must also be defined somewhere:

```
void Stack::push(char c)
{
    if (top == max_size) throw Overflow();
    v[top] = c;
    top = top + 1;
}

char Stack::pop()
{
    if (top == 0) throw Underflow();
    top = top - 1;
    return v[top];
}
```

Types such as ***complex*** and ***Stack*** are called *concrete types*, in contrast to *abstract types*, where the interface more completely insulates a user from implementation details.

2.5.4 Abstract Types

One property was lost in the transition from ***Stack*** as a ''fake type'' implemented by a module (§2.5.1) to a proper type (§2.5.3). The representation is not decoupled from the user interface; rather, it is a part of what would be included in a program fragment using ***Stack***s. The representation is private, and therefore accessible only through the member functions, but it is present. If it changes in any significant way, a user must recompile. This is the price to pay for having concrete types behave exactly like built-in types. In particular, we cannot have genuine local variables of a type without knowing the size of the type's representation.

For types that don't change often, and where local variables provide much-needed clarity and efficiency, this is acceptable and often ideal. However, if we want to completely isolate users of a

stack from changes to its implementation, this last ***Stack*** is insufficient. Then, the solution is to decouple the interface from the representation and give up genuine local variables.

First, we define the interface:

```
class Stack {
public:
        class Underflow { };        // used as exception
        class Overflow { };         // used as exception

        virtual void push(char c) = 0;
        virtual char pop() = 0;
};
```

The word ***virtual*** means ''may be redefined later in a class derived from this one'' in Simula and C++. A class derived from ***Stack*** provides an implementation for the ***Stack*** interface. The curious ***=0*** syntax says that some class derived from ***Stack*** *must* define the function. Thus, this ***Stack*** can serve as the interface to any class that implements its ***push***`()` and ***pop***`()` functions.

This ***Stack*** could be used like this:

```
void f(Stack& s_ref)
{
        s_ref.push('c');
        if (s_ref.pop() != 'c') throw Bad_pop();
}
```

Note how ***f***`()` uses the ***Stack*** interface in complete ignorance of implementation details. A class that provides the interface to a variety of other classes is often called a *polymorphic type*.

Not surprisingly, the implementation could consist of everything from the concrete class ***Stack*** that we left out of the interface ***Stack***:

```
class Array_stack : public Stack {      // Array_stack implements Stack
        char* p;
        int max_size;
        int top;
public:
        Array_stack(int s);
        ~Array_stack();

        void push(char c);
        char pop();
};
```

The ''***:public***'' can be read as ''is derived from,'' ''implements,'' and ''is a subtype of.''

For a function like ***f***`()` to use a ***Stack*** in complete ignorance of implementation details, some other function will have to make an object on which it can operate. For example:

```
void g()
{
        Array_stack as(200);
        f(as);
}
```

Since *f*( ) doesn't know about ***Array_stack***s but only knows the ***Stack*** interface, it will work just as well for a different implementation of a ***Stack***. For example:

```
class List_stack : public Stack {       // List_stack implements Stack
    list<char> lc;                      // (standard library) list of characters (§3.7.3)
public:
    List_stack() { }

    void push(char c) { lc.push_front(c); }
    char pop();
};

char List_stack::pop()
{
    char x = lc.front();                // get first element
    lc.pop_front();                     // remove first element
    return x;
}
```

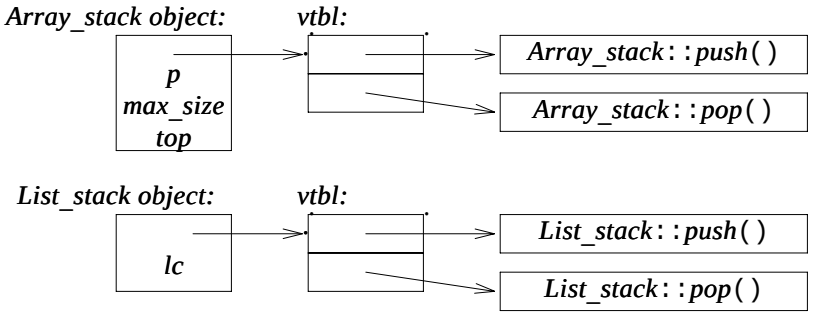
Here, the representation is a list of characters. The ***lc.push_front(c)*** adds ***c*** as the first element of ***lc***, the call ***lc.pop_front()*** removes the first element, and ***lc.front()*** denotes ***lc***'s first element.

A function can create a ***List_stack*** and have *f*( ) use it:

```
void h()
{
    List_stack ls;
    f(ls);
}
```

2.5.5 Virtual Functions

How is the call ***s_ref.pop()*** in *f*( ) resolved to the right function definition? When *f*( ) is called from *h*( ), ***List_stack::pop()*** must be called. When *f*( ) is called from *g*( ), ***Array_stack::pop()*** must be called. To achieve this resolution, a ***Stack*** object must contain information to indicate the function to be called at run-time. A common implementation technique is for the compiler to convert the name of a ***virtual*** function into an index into a table of pointers to functions. That table is usually called ''a virtual function table'' or simply, a ***vtbl***. Each class with virtual functions has its own ***vtbl*** identifying its virtual functions. This can be represented graphically like this:



The functions in the ***vtbl*** allow the object to be used correctly even when the size of the object and the layout of its data are unknown to the caller. All the caller needs to know is the location of the ***vtbl*** in a ***Stack*** and the index used for each virtual function. This virtual call mechanism can be made essentially as efficient as the ''normal function call'' mechanism. Its space overhead is one pointer in each object of a class with virtual functions plus one ***vtbl*** for each such class.

2.6 Object-Oriented Programming

Data abstraction is fundamental to good design and will remain a focus of design throughout this book. However, user-defined types by themselves are not flexible enough to serve our needs. This section first demonstrates a problem with simple user-defined data types and then shows how to overcome that problem by using class hierarchies.

2.6.1 Problems with Concrete Types

A concrete type, like a ''fake type'' defined through a module, defines a sort of black box. Once the black box has been defined, it does not really interact with the rest of the program. There is no way of adapting it to new uses except by modifying its definition. This situation can be ideal, but it can also lead to severe inflexibility. Consider defining a type ***Shape*** for use in a graphics system. Assume for the moment that the system has to support circles, triangles, and squares. Assume also that we have

```
class Point{ /* ... */ };
class Color{ /* ... */ };
```

The `/*` and `*/` specify the beginning and end, respectively, of a comment. This comment notation can be used for multi-line comments and comments that end before the end of a line.

We might define a shape like this:

```
enum Kind { circle, triangle, square };    // enumeration (§4.8)

class Shape {
      Kind k;          // type field
      Point center;
      Color col;
      // ...

public:
      void draw();
      void rotate(int);
      // ...
};
```

The ''type field'' ***k*** is necessary to allow operations such as ***draw()*** and ***rotate()*** to determine what kind of shape they are dealing with (in a Pascal-like language, one might use a variant record with tag ***k***). The function ***draw()*** might be defined like this:

```
void Shape::draw()
{
    switch (k) {
    case circle:
        // draw a circle
        break;
    case triangle:
        // draw a triangle
        break;
    case square:
        // draw a square
        break;
    }
}
```

This is a mess. Functions such as ***draw()*** must ''know about'' all the kinds of shapes there are. Therefore, the code for any such function grows each time a new shape is added to the system. If we define a new shape, every operation on a shape must be examined and (possibly) modified. We are not able to add a new shape to a system unless we have access to the source code for every operation. Because adding a new shape involves ''touching'' the code of every important operation on shapes, doing so requires great skill and potentially introduces bugs into the code that handles other (older) shapes. The choice of representation of particular shapes can get severely cramped by the requirement that (at least some of) their representation must fit into the typically fixed-sized framework presented by the definition of the general type ***Shape***.

2.6.2 Class Hierarchies

The problem is that there is no distinction between the general properties of every shape (that is, a shape has a color, it can be drawn, etc.) and the properties of a specific kind of shape (a circle is a shape that has a radius, is drawn by a circle-drawing function, etc.). Expressing this distinction and taking advantage of it defines object-oriented programming. Languages with constructs that allow this distinction to be expressed and used support object-oriented programming. Other languages don't.

The inheritance mechanism (borrowed for C++ from Simula) provides a solution. First, we specify a class that defines the general properties of all shapes:

```
class Shape {
    Point center;
    Color col;
    // ...
public:
    Point where() { return center; }
    void move(Point to) { center = to; /* ... */ draw(); }

    virtual void draw() = 0;
    virtual void rotate(int angle) = 0;
    // ...
};
```

As in the abstract type ***Stack*** in §2.5.4, the functions for which the calling interface can be defined – but where the implementation cannot be defined yet – are ***virtual***. In particular, the functions ***draw***() and ***rotate***() can be defined only for specific shapes, so they are declared ***virtual***.

Given this definition, we can write general functions manipulating vectors of pointers to shapes:

```
void rotate_all(vector<Shape*>& v, int angle) // rotate v's elements angle degrees
{
    for (int i = 0; i<v.size(); ++i) v[i]->rotate(angle);
}
```

To define a particular shape, we must say that it is a shape and specify its particular properties (including the virtual functions):

```
class Circle : public Shape {
    int radius;
public:
    void draw() { /* ... */ }
    void rotate(int) {}   // yes, the null function
};
```

In C++, class ***Circle*** is said to be *derived* from class ***Shape***, and class ***Shape*** is said to be a *base* of class ***Circle***. An alternative terminology calls ***Circle*** and ***Shape*** subclass and superclass, respectively. The derived class is said to inherit members from its base class, so the use of base and derived classes is commonly referred to as *inheritance*.

The programming paradigm is:

> *Decide which classes you want;*
> *provide a full set of operations for each class;*
> *make commonality explicit by using inheritance.*

Where there is no such commonality, data abstraction suffices. The amount of commonality between types that can be exploited by using inheritance and virtual functions is the litmus test of the applicability of object-oriented programming to a problem. In some areas, such as interactive graphics, there is clearly enormous scope for object-oriented programming. In other areas, such as classical arithmetic types and computations based on them, there appears to be hardly any scope for more than data abstraction, and the facilities needed for the support of object-oriented programming seem unnecessary.

Finding commonality among types in a system is not a trivial process. The amount of commonality to be exploited is affected by the way the system is designed. When a system is designed – and even when the requirements for the system are written – commonality must be actively sought. Classes can be designed specifically as building blocks for other types, and existing classes can be examined to see if they exhibit similarities that can be exploited in a common base class.

For attempts to explain what object-oriented programming is without recourse to specific programming language constructs, see [Kerr,1987] and [Booch,1994] in §23.6.

Class hierarchies and abstract classes (§2.5.4) complement each other instead of being mutually exclusive (§12.5). In general, the paradigms listed here tend to be complementary and often

mutually supportive. For example, classes and modules contain functions, while modules contain classes and functions. The experienced designer applies a variety of paradigms as need dictates.

2.7 Generic Programming

Someone who wants a stack is unlikely always to want a stack of characters. A stack is a general concept, independent of the notion of a character. Consequently, it ought to be represented independently.

More generally, if an algorithm can be expressed independently of representation details and if it can be done so affordably and without logical contortions, it ought to be done so.

The programming paradigm is:

> *Decide which algorithms you want;*
> *parameterize them so that they work for*
> *a variety of suitable types and data structures.*

2.7.1 Containers

We can generalize a stack-of-characters type to a stack-of-anything type by making it a *template* and replacing the specific type ***char*** with a template parameter. For example:

```
template<class T> class Stack {
        T* v;
        int max_size;
        int top;
public:
        class Underflow { };
        class Overflow { };

        Stack(int s);    // constructor
        ~Stack();        // destructor

        void push(T);
        T pop();
};
```

The ***template<class T>*** prefix makes ***T*** a parameter of the declaration it prefixes.

The member functions might be defined similarly:

```
template<class T> void Stack<T>::push(T c)
{
        if (top == max_size) throw Overflow();
        v[top] = c;
        top = top + 1;
}
```

```
template<class T> T Stack<T>::pop()
{
    if (top == 0) throw Underflow();
    top = top - 1;
    return v[top];
}
```

Given these definitions, we can use stacks like this:

```
Stack<char> sc(200);          // stack of 200 characters
Stack<complex> scplx(30);     // stack of 30 complex numbers
Stack< list<int> > sli(45);   // stack of 45 lists of integers

void f()
{
    sc.push('c');
    if (sc.pop() != 'c') throw Bad_pop();

    scplx.push(complex(1,2));
    if (scplx.pop() != complex(1,2)) throw Bad_pop();
}
```

Similarly, we can define lists, vectors, maps (that is, associative arrays), etc., as templates. A class holding a collection of elements of some type is commonly called a *container class*, or simply a *container*.

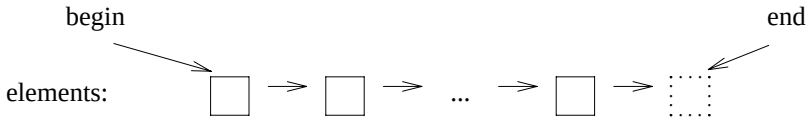
Templates are a compile-time mechanism so that their use incurs no run-time overhead compared to ''hand-written code.''

2.7.2 Generic Algorithms

The C++ standard library provides a variety of containers, and users can write their own (Chapter 3, Chapter 17, Chapter 18). Thus, we find that we can apply the generic programming paradigm once more to parameterize algorithms by containers. For example, we want to sort, copy, and search ***vector***s, ***list***s, and arrays without having to write ***sort***(), ***copy***(), and ***search***() functions for each container. We also don't want to convert to a specific data structure accepted by a single sort function. Therefore, we must find a generalized way of defining our containers that allows us to manipulate one without knowing exactly which kind of container it is.

One approach, the approach taken for the containers and non-numerical algorithms in the C++ standard library (§3.8, Chapter 18) is to focus on the notion of a sequence and manipulate sequences through iterators.

Here is a graphical representation of the notion of a sequence:



A sequence has a beginning and an end. An iterator refers to an element, and provides an operation that makes the iterator refer to the next element of the sequence. The end of a sequence is an

iterator that refers one beyond the last element of the sequence. The physical representation of ''the end'' may be a sentinel element, but it doesn't have to be. In fact, the point is that this notion of sequences covers a wide variety of representations, including lists and arrays.

We need some standard notation for operations such as ''access an element through an iterator'' and ''make the iterator refer to the next element.'' The obvious choices (once you get the idea) are to use the dereference operator * to mean ''access an element through an iterator'' and the increment operator ++ to mean ''make the iterator refer to the next element.''

Given that, we can write code like this:

```
template<class In, class Out> void copy(In from, In too_far, Out to)
{
    while (from != too_far) {
        *to = *from;    // copy element pointed to
        ++to;           // next output
        ++from;         // next input
    }
}
```

This copies any container for which we can define iterators with the right syntax and semantics.

C++'s built-in, low-level array and pointer types have the right operations for that, so we can write

```
char vc1[200]; // array of 200 characters
char vc2[500]; // array of 500 characters

void f()
{
    copy(&vc1[0],&vc1[200],&vc2[0]);
}
```

This copies *vc1* from its first element until its last into *vc2* starting at *vc2*'s first element.

All standard library containers (§16.3, Chapter 17) support this notion of iterators and sequences.

Two template parameters ***In*** and ***Out*** are used to indicate the types of the source and the target instead of a single argument. This was done because we often want to copy from one kind of container into another. For example:

```
complex ac[200];

void g(vector<complex>& vc, list<complex>& lc)
{
    copy(&ac[0],&ac[200],lc.begin());
    copy(lc.begin(),lc.end(),vc.begin());
}
```

This copies the array to the ***list*** and the ***list*** to the ***vector***. For a standard container, ***begin()*** is an iterator pointing to the first element.

2.8 Postscript

No programming language is perfect. Fortunately, a programming language does not have to be perfect to be a good tool for building great systems. In fact, a general-purpose programming language cannot be perfect for all of the many tasks to which it is put. What is perfect for one task is often seriously flawed for another because perfection in one area implies specialization. Thus, C++ was designed to be a good tool for building a wide variety of systems and to allow a wide variety of ideas to be expressed directly.

Not everything can be expressed directly using the built-in features of a language. In fact, that isn't even the ideal. Language features exist to support a variety of programming styles and techniques. Consequently, the task of learning a language should focus on mastering the native and natural styles for that language – not on the understanding of every little detail of all the language features.

In practical programming, there is little advantage in knowing the most obscure language features or for using the largest number of features. A single language feature in isolation is of little interest. Only in the context provided by techniques and by other features does the feature acquire meaning and interest. Thus, when reading the following chapters, please remember that the real purpose of examining the details of C++ is to be able to use them in concert to support good programming style in the context of sound designs.

2.9 Advice

[1] Don't panic! All will become clear in time; §2.1.
[2] You don't have to know every detail of C++ to write good programs; §1.7.
[3] Focus on programming techniques, not on language features; §2.1.

.