Block-1

1.1 Learning Objective

After going through this unit, the learner will able to

- Define the Object-Oriented Programming
- Understand the features of Object-Oriented Programming
- Learn about namespaces
- Define the Class and Object
- Learn about the Object terminology
- Learn about modular Programming

1.2 Introduction to Object-Oriented Programming

Modern software applications are intricate, dynamic and complex. The number of lines of code can exceed the hundreds of thousands or millions. These applications evolve over considerable time. Some applications take years of programming effort and more years to mature. Creating such applications involves many developers with different levels of expertise. These software projects consist of smaller stand alone and testable sub-projects; sub-projects that are transferrable, practical, upgradeable and possibly even usable within other projects. The principles of software engineering suggest that each component should be a highly cohesive element and that the collection of components should be loosely coupled. Object-oriented languages provide the tools for implementing these general principles.

C++ is an object-oriented programming language specifically designed to provide a simple, comprehensive feature set for programming modern applications without any loss in performance capabilities of a 'close to the metal' language. C++ combines the efficiencies of the C language with the object-oriented features necessary for the development of large applications.

Mastering Complexity

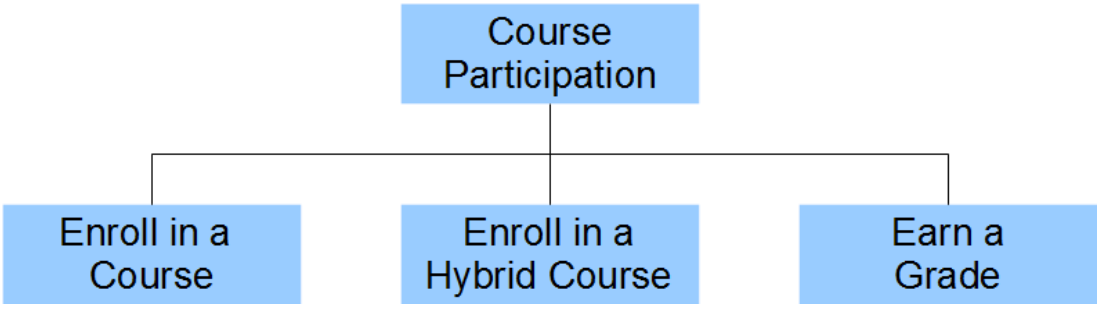
Large applications are complex. We master their complexity by identifying the most important features of the problem domain. In very general terms, we express the features in terms of *data*

and *activities*. We identify the data objects and the activities on those objects as complementary tasks.

Consider a course enrollment system for a program in a college or university. Each participant

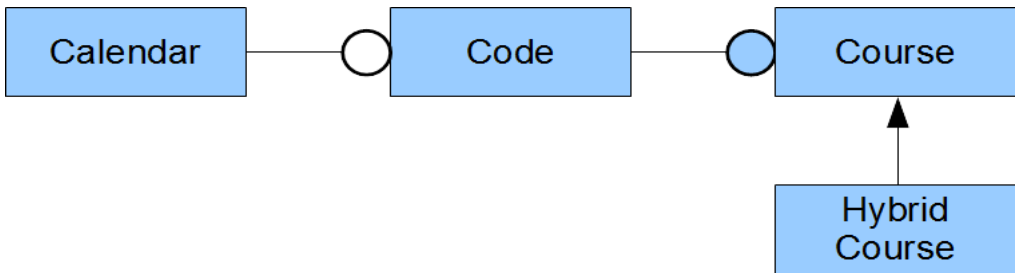
- enrolls in several face-to-face courses
- enrolls in several hybrid course
- earns a grade in each course

The following structure diagram shows the set of activities.



If we switch our attention from these activities to the objects involved, we identify a Course and a Hybrid Course. Focusing on a Course, we observe that it has a Course Code. We look up the Code in the Calendar to determine when the Course is being offered.

We say that a Course *has a* Code and that a Code *uses* the Calendar to determine its availability. The diagram below shows these relationships between the objects in the problem domain. The connectors identify the types of relationships between the objects.



In switching our attention from the activities in the structure chart to the objects in the relationship diagram we switch from a procedural description of the problem to an object-oriented description.

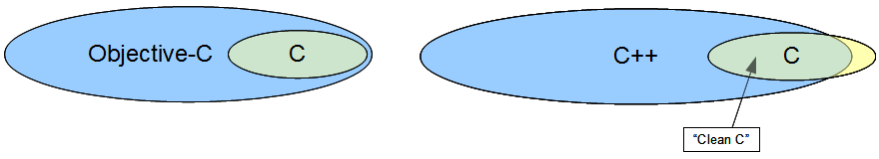
These two distinct approaches to complexity date at least as back as the ancient Greeks. Heraclitus viewed the world in terms of process, while Democritus viewed the world in terms of discrete atoms.

Learning to divide a complex problem into objects and to identify the relationships amongst the objects is the subject matter of a course on system analysis and design. The material covered in this text introduces some of the principal concepts central to analysis and design along with the C++ syntax for implementing these concepts in code.

Programming Languages

Eric Levenez maintains a web page that lists the major programming languages throughout the world. TIOBE Software tracks the most popular ones and long-term trends based on world-wide availability of software engineers as calculated from Google, Yahoo!, and MSN search engines. Many of these languages support object orientation.

Java, C, C++ and Objective-C are currently the four most popular languages. Each is an *imperative* language; that is, a language that specifies each step necessary to reach a desired state. These languages have much syntax in common: Objective-C is a strict superset of C, C++ contains almost all of C as a subset, Java syntax is C-like, but not a superset of C.



The distinguishing features are:

- C has no object-oriented support. C leaves us no choice but to design our solutions in terms of activity-based logic.

- Objective-C is hybrid. It augments the C language with object-oriented features. Objective-C lets us build our solutions partly from activities and partly from objects. The main function in any Objective-C program is a C function, which is not object-oriented. Objective-C stresses run-time logic.
- C++, like Objective-C, is hybrid. It augments C with object-oriented features. C++ lets us build our solutions partly from activities and partly from objects. The main function in any C++ program is a C function, which is not object-oriented. C++, as opposed to Objective-C, stresses compile-time logic.
- Java is purely object-oriented. It excludes all non-object-oriented features. Java leaves us no choice but to design our solutions using an object-oriented logic.

Features of C++

Learning object-oriented programming using C++ has several advantages for a student familiar with C. C++ is

- nearly a perfect superset of C
- a multi-paradigm language
    - procedural (can focus on distinct activities)
    - object-oriented (can focus on distinct objects)
- realistic, efficient, and flexible enough for demanding projects
    - large applications
    - game programming
    - operating systems
- clean enough for presenting basic concepts
- comprehensive enough for presenting advanced concepts

Type Safety

Type safety in central to C++

A type-safe language traps syntax errors at compile-time, diminishing the amount of buggy code that escapes to the client. The compiler uses type rules to check syntax and generates errors or warnings if any rule is violated.

C compilers are more tolerant of type errors than C++ compilers. For example, a C compiler will accept the following code, which may cause a segmentation fault at run-time

```c
#include <stdio.h>
void foo();

int main(void)
{
    foo(-25);
}
void foo(char x[])
{
    printf("%s", x); /* ERROR */
}
```

```
Segmentation Fault (coredump)
```

The prototype for foo() instructs the compiler to omit checking for argument/parameter type mismatches. The argument in the function call is an int of negative value (-25) and the type received in the parameter is the address of a char array. Since the parameter's value is an invalid address, printing from that address causes a segmentation fault at run-time, but no error at compile-time.

We can fix this easily. If we include the parameter type in the prototype as shown below, the compiler will check for an argument/parameter type mismatch and issue an error message like that shown on the right:

```c
#include <stdio.h>
void foo(char x[]);

int main(void)
{
    foo(-25);
}
void foo(char x[])
{
    printf("%s", x);
}
```

```
Function argument assignment between
types "char*" and "int" is not allowed.
```

Bjarne Stroustrup, in creating the C++ language, decided to close this loophole. He mandated that all prototypes list their parameter types, which forces all C++ compilers to check for argument/parameter type mismatches at compile-time.

Namespaces

In applications written simultaneously by many developers, chances are high that different developers will use the same identifiers for different variables in the application. If so, once they assemble their code, naming conflicts will arise. We avoid such conflicts by developing each part of an application within its own namespace and identifying its variables relative to that namespace.

A *namespace* is a scope for the entities that it encloses. Scoping rules prevent identifier conflicts across different namespaces.

We define a namespace as follows

```
namespace identifier {

}
```

The identifier after the namespace keyword is the name of the scope. The pair of braces encloses and defines the scope.

For example, to define x in two separate namespaces (english and french), we write

```
namespace english {

    int x = 2;
    // ...

}

namespace french {

    int x = 3;
    // ...

}
```

To access a variable defined within a namespace, we precede its identifier with the namespace's identifier and separate them with a double colon (::). We call this double colon the *scope resolution operator*.

For example, to increment the x in namespace english and to decrement the x in namespace french, we write

```
    english::x++;
    french::x--;
```

Each prefix uniquely identifies each variable's namespace.

Namespaces hide entities. To expose an identifier to the current namespace, we insert the using declaration before referring to the identifier.

For example, to expose one of the x's to the current namespace, we write:

```
using french::x;
```

After which, we can simply write:

```
    x++;  // increments french::x but not english::x
```

To expose all of the identifiers within a namespace, we insert the using directive before referring to any of them.

For example, to expose all of the identifiers within namespace english, we write:

```
using namespace english;
```

Afterwards, we can write:

```
  x++; // increments english::x but not french::x
```

Exposing a single identifier or a complete namespace simply adds the identifier(s) to the hosting namespace.

From C to C++ Syntax

To compare C++ with C syntax, consider a program that displays the following phrase on the standard output device

```
Welcome to Object-Oriented
```

C - procedural code

The C source code for displaying this phrase is

```
// A Language for Complex Applications
//  welcome.c
//
// To compile on linux: gcc welcome.c
// To run compiled code: a.out
//
// To compile on windows: cl welcome.c
// To run compiled code: welcome
//

#include <stdio.h>

int main(void)
{
    printf("Welcome to Object-Oriented\n");
}
```

The two functions - main() and printf() - identify activities. These identifiers share the global namespace.

C++ - procedural code

The procedural C++ source code for displaying the phrase is

```
// A Language for Complex Applications
// welcome.cpp
//
// To compile on linux:   g++ welcome.cpp
//  To run compiled code: a.out
//
// To compile on windows:   cl welcome.cpp
//  To run compiled code: welcome
//

#include <cstdio>
using namespace std;

int main ( ) {
    printf("Welcome to Object-Oriented\n");
}
```

<cstdio> is the C++ version of the C header file <stdio.h>. This header file declares the prototype for printf() within the std namespace. std stands for standard. The file extension for any C++ source code is .cpp.

The directive

```
using namespace std;
```

exposes all of the identifiers declared within the std namespace to the global namespace. The libraries of standard C++ declare most of their identifiers within the std namespace.

C++ - hybrid code

The object-oriented C++ source code for displaying our welcome phrase is

```
// A Language for Complex Applications
// welcome.cpp
//
// To compile on linux:  g++ welcome.cpp
// To run compiled code: a.out
//
// To compile on windows:   cl welcome.cpp
//  To run compiled code: welcome
//

#include <iostream>
using namespace std;

int main ( ) {
    cout << "Welcome to Object-Oriented" << endl;
}
```

The object-oriented syntax consists of:

1. The directive

   ```
   #include <iostream>
   ```

   inserts the <iostream> header file into the source code. The <iostream> library provides access to the standard input and output objects.

2. The object

   ```
   cout
   ```

   represents the standard output device.

3. The operator

   ```
   <<
   ```

   inserts whatever is on its right side into whatever is on its left side.

4. The manipulator

   ```
   endl
   ```

   represents an end of line character along with a flushing of the output buffer.

Note the absence of a formatting string. The cout object handles the output formatting itself. That is, the complete statement

```
cout << "Welcome to Object-Oriented" << endl;
```

inserts into the standard output stream the string "Welcome to Object-Oriented" followed by a newline character and a flushing of the output buffer.

First Input and Output Example

The following object-oriented program accepts an integer value from standard input and displays that value on standard output:

```
// Input Output Objects
// inputOutput.cpp
//
// To compile on linux: g++ inputOutput.cpp
// To run compiled code: a.out
//
// To compile on windows:   cl welcome.cpp
// To run compiled code: welcome
//

#include <iostream>
using namespace std;

int main() {
    int i;

    cout << "Enter an integer : ";
    cin >> i;
    cout << "You entered " << i << endl;
}
```

```
Enter an integer : 65

You entered 65
```

The object-oriented input statement includes:

- The object

  `cin`

  represents the standard input device.

- The extraction operator

`>>`

extracts the data identified on its right side from the object on its left-hand side. Note the absence of a formatting string. The cin object handles the input formatting itself. That is, the complete statement

```
cin >> i;
```

extracts an integer value from the input stream and stores that value in the variable named i The type of the variable i defines the rule for converting the text characters in the input stream into byte data in memory. Note the absence of the address of operator on i and the absence of the conversion specifier, each of which is present in the C language.

1.3 Object Terminology

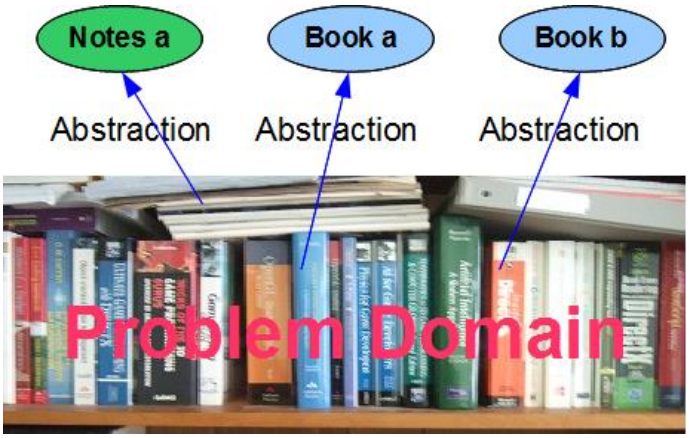
Object-oriented programming is reflects the way in which we manage our day-to-day tasks. Professor Miller of Princeton University demonstrated that most of us can only comprehend about seven chunks of information at a time. As children, we learn to play with small sets of chunks at an early age. As we grow, we learn to break down each problem that we face into a set of manageable chunks.

A chunk in object-oriented programming is called an *object*. Object-oriented languages refer to the shared structure of a set of similar objects as their *class*. This structure includes the structure of the data in the similar objects as well as the logic that works on that data.

This chapter defines an object and a class and introduces the concepts of encapsulation, inheritance and polymorphism. Subsequent chapters elaborate on these concepts in detail.

Abstraction

An object-oriented solution to a programming problem consists of components represented as objects. We call the process of designing an object-oriented solution *abstraction*. In this process, we distinguish the most important feature and hide the less important details within the objects themselves.

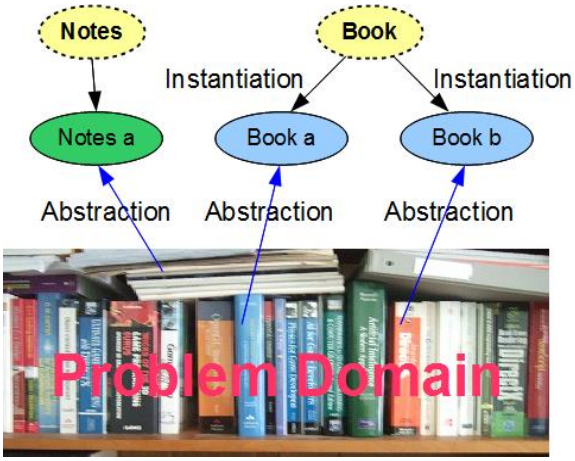


Each object has a crisp conceptual boundary and acts in ways appropriate to itself. Compare a book with a set of notes. A book has pages that are bound and can be flipped. The page order is fixed and cannot be changed. A set of notes consists of loose pages that can be rearranged in any order. We represent the book and the set of notes as objects, but each object has a different structure.

The cout and cin objects introduced in the preceding chapter have different structures. cout represents the standard output device, which may be a monitor or a printer. The abstraction - the standard output device - is simple and distinct. Internally, the cout object is quite complex. On the other hand, cin represents the standard input device, which may be a keyboard, a tablet or a touch screen. The abstraction - the standard input device - is also simple and distinct. Internally, the cin object is also quite complex.

Classes

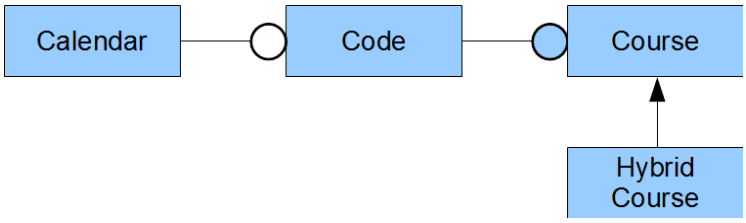
We describe the structure of similar objects in terms of a *class*. Objects of the same class have the same structure, but possibly different states. The variables that describe their states generally have different values, but the variable types are identical. For example, all of the books in the Figure above have a title and an author, but each book has a different title and a different author.



We say that each object is an *instance* of its class.

UML

The Unified Modelling Language (UML) is a general-purpose modeling language developed that describes systems of objects and relationships between classes. This language defines standard symbols for classes and the relationships between them. The connectors shown in the relationship diagram below are UML connectors. We use these symbols in this text.

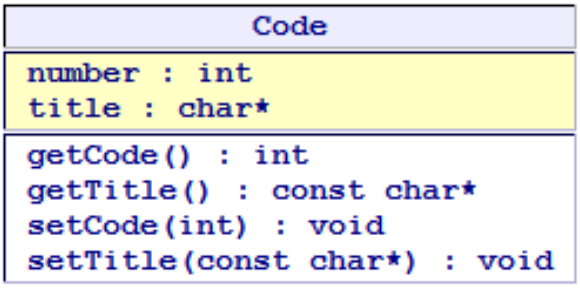


The Class Diagram

The primary graphic in UML is the *class diagram*: a rectangular box with three compartments:

1. the upper compartment identifies the class by its name
2. the middle compartment identifies the names and types of its attributes
3. the lower compartment identifies the names, return types and parameter types of its operations

For example,



The naming conventions include:

- begin each class name with an upper case letter
- begin each member name with a lower case letter
- use camel case throughout all names - capitalize the first letter of every word after the first word

Terminology

UML uses the terms attributes and operations. Each object-oriented language uses its own terms. Equivalent terms are:

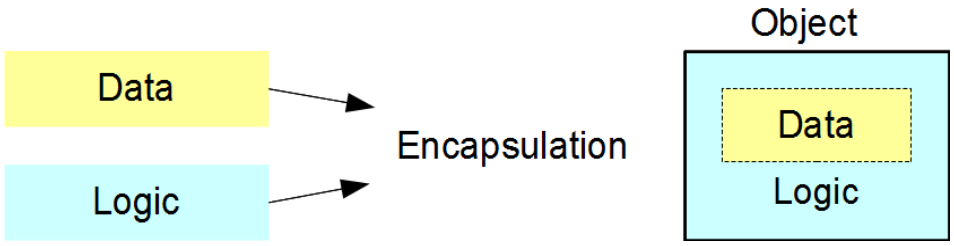
- attributes (UML) -> fields, data members (C++), properties
- operations (UML) -> methods (Java), procedures, messages, member functions (C++)

The C++ language standard uses the terms *data members* and *member functions* exclusively.

Encapsulation

Encapsulation is the primary concept of object-oriented programming. It refers to the integration of data and logic within a class' implementation and the crisp interface between any client and that implementation. In other words, encapsulation is a technique that supports high cohesion within a class and low coupling between the class' implementation and any one of its clients.

The class definition declares the variables that store each object's data and the prototypes of the functions that contain the logic that operates on that data. Clients access objects through calls to these functions without knowlegde or any need to know the data data stored within the objects or the logic that manipulates that data. The function prototypes provide the interface.



Encapsulation shileds the complex details of an object's implementation from its crisp external representation. Consider the following statement from the preceding chapter:

```
cout << "Welcome to Object-Oriented";
```

cout refers to the standard output object. Its class defines how to store any data in memory and how to control the processes that work with that data. The << operator copies the string to the output object without exposing any of the implementation details. As clients, we only see the interface that manages the output process.

A well-encapsulated class hides all of the implementation details within itself. The client does not see the data that the class' object stores within itself or the logic that it uses to manage its internal data. The client only sees a clean and simple interface to the object.

As long as the classes in a programming solution are well-encapsulated, any programmer can upgrade the internal structure of any object developed by another programmer without changing any client code.

Inheritance and Polymorphism

Two other concepts in object-oriented languages are prominent in our study of relationships between classes:

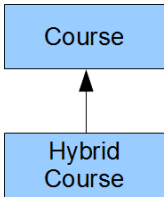
- Inheritance - one class inherits the structure of another class
- Polymorphism - a single interface provides multiple implementations

These are special cases of encapsulation in the sense that they distinquish interface and implementation to produce highly cohesive objects that support minimal coupling to their clients.

Inheritance

Inheritance relates classes that share the same structure. In the Figure below, the Hybrid Course class inherits the entire structure of the Course class. We say that the hybrid course '*is-a-kind-of*'

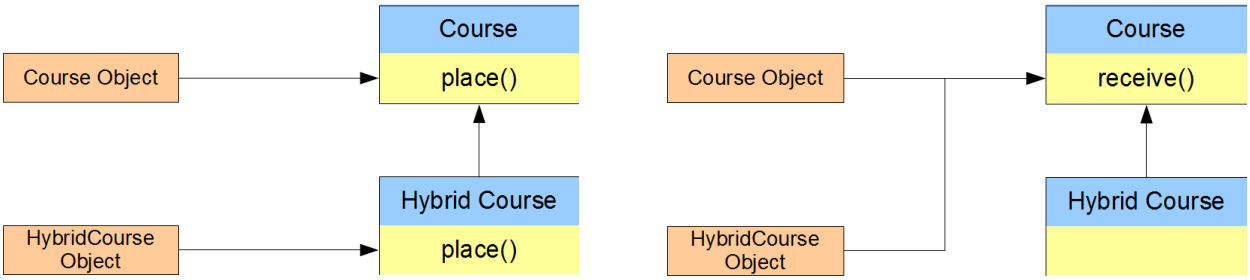
course and depict the inheritance relationship using an arrow drawn from the more specialized class to the more general class:



Polymorphism

Polymorphism associates the implementation appropriate to an object based on its type. In the Figure below, the HybridCourse object uses a different mode of delivery than the Course object, but the same assessments. Note that both objects belong to the same hierarchy: both are Course objects.

A mode() query on a Course object reports a different result than a mode() query on a Hybrid Course object. On the other hand, an assessments() query on a Course object reports the same result as on the HybridCourse object. Duplicating identical code is avoided under polymorphism.



The Three Muskateers

Encapsulation, inheritance and polymorphism are the three foundational concepts of any object-oriented programming language.

1.4 Modular Programming

Modular programming implements modular designs and is supported by procedural and object-oriented languages. A modular design consists of a set of *modules*, which were developed and tested separately. The C programming language supports modular design through library modules composed of functions. The stdio module provides input and output support and hides its implementation details; typically, the implementation for scanf() and printf() ships in binary

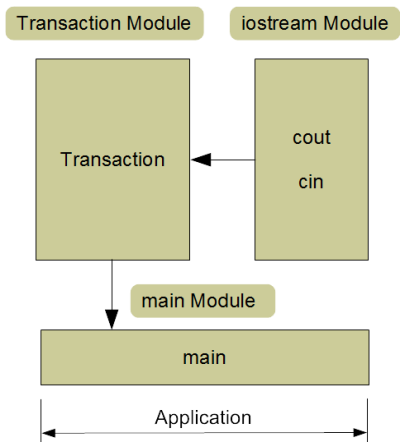
form with the compiler. The stdio.h header file provides the interface, which is all that we need to complete our source code.

This chapter describes how to create a module in C++, compile the source code for each module separately and link the compiled code into a single executable binary. The chapter concludes with an example of a unit test on a module.

Modules

A well-designed module is a highly cohesive unit that can be loosely coupled to other modules. It addresses one aspect of a programming solution and hides as much detail as practically possible. A compiler translates the source code for a module independently of the source code for other modules into its own unit of binary code.

Consider the Transaction program illustrated below. The main module accesses the Transaction module. The Transaction module accesses the iostream module. The Transaction module contains the definitions of the transaction functions used by the program. The iostream module contains the definitions of the cout and cin objects used by the program.



To translating the source code of any module the compiler needs information identifying the names used in that modules but defined outside the module. To enable this, we store C++ source code for each module in two separate files:

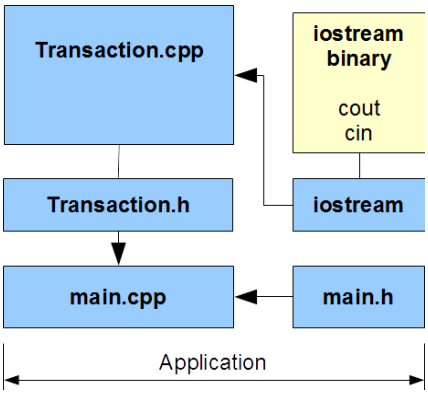
- a header file - contains the class definitions and the function prototypes
- an implementation file - contains the instructions that define the logic within the functions

The file extension .h (or .hpp) identifies the header file. The file extension .cpp identifies the implementation file.

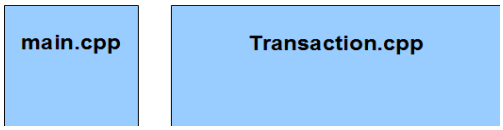
The names of the header files for the standard C++ libraries do not include a file extension (consider for example, the <iostream> header file for the cout and cin objects).

Example

The implementation file for the main module in the Transaction program above includes the header files for the itself (main.h) and for the Transaction module (Transaction.h). The implementation file for the Transaction module includes the header file for the iostream module. Each implementation file DOES NOT include any other implementation file.



We compile each implementation (*.cpp) file separately and only once. We do not compile header (*.h) files.

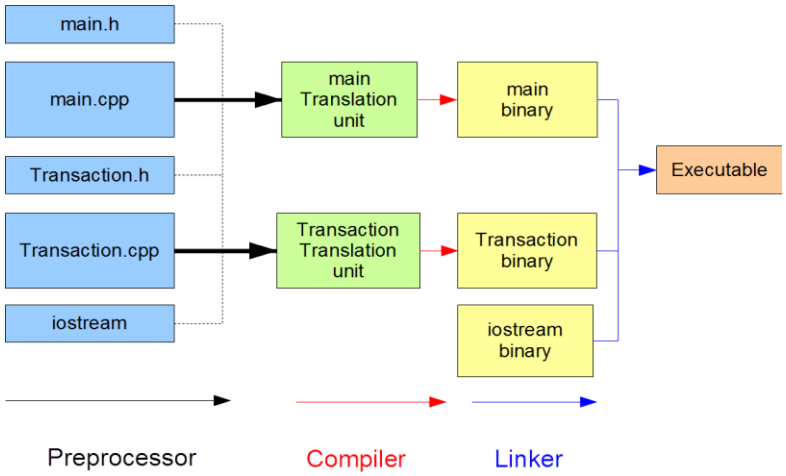


The compiled version of iostream's implementation file is part of the system library.

Stages of Compilation

Comprehensive compilation consists of three independent but sequential stages (as shown in the figure below):

1. Preprocessor - interprets all directives and creates a single translation unit for the compiler - (#include inserts the contents of the header files), (#define substitutes all macros)
2. Compiler - compiles each translation unit separately and creates an independent binary version
3. Linker - assembles the various binary units along with the system binaries to create one complete executable binary



A Modular Example

Consider a trivial accounting application that accepts journal transaction data from the standard input device and displays that data on the standard output device. The application does not perform any intermediate calculation.

The design consists of two modules:

- Main - supervises the processing of all transactions
- Transaction - defines the input and output logic for a single transaction

Transaction Module

Let us define a structure for a single transaction

- Transaction - holds the information for one transaction in memory

and two global functions

- enter() - accepts transaction data from the standard input device
- display() - displays transaction data on the standard output device

Transaction.h

The header file for our Transaction module defines our Transaction type and declares the prototypes for our two functions:

```
// Modular Example
// Transaction.h

struct Transaction {
    int acct;      // account number
    char type;     // credit 'c' debit 'd'
    double amount; // transaction amount
};

void enter(struct Transaction* tr);
void display(const struct Transaction* tr);
```

Note the UML naming convention and the extension on the name of the header file.

Transaction.cpp

The implementation file for our Transaction module defines the logic within our two functions. It includes the system header file for access to the cout and cin objects and the header file for access to the Transaction type.

```
// Modular Example
// Transaction.cpp

#include <iostream> // for cout, cin
using namespace std;
#include "Transaction.h"  // for Transaction

// prompts for and accepts Transaction data
//
void enter(struct Transaction* tr) {

    cout << "Enter the account number : ";
    cin  >> tr->acct;
    cout << "Enter the account type (d debit, c credit) : ";
    cin  >> tr->type;
    cout << "Enter the account amount : ";
    cin  >> tr->amount;
}

// displays Transaction data
//
void display(const struct Transaction* tr) {

    cout << "Account " << tr->acct;
    cout << ((tr->type == 'd') ? " Debit $" : " Credit $") << tr->amount;
    cout << endl;
}
```

Note the .cpp extension on the name of this implementation file

Main Module

The main module defines a Transaction object and accepts input and displays data for each of three transactions.

main.h

The header file for our Main module #defines the number of transactions:

```
// Modular Example
// main.h

#define NO_TRANSACTIONS 3
```

main.cpp

The implementation file for our Main module defines the main() function. We #include the header file for the definition of the Transaction type:

```
// Modular Example
// main.cpp

#include "main.h"
#include "Transaction.h"

int main() {
    int i;
    struct Transaction tr;

    for (i = 0; i < NO_TRANSACTIONS; i++) {
        enter(&tr);
        display(&tr);
    }
}
```

Command Line Compilation

Linux

To compile our application on a Linux platform at the command line, we enter the following

```
g++ -o accounting main.cpp Transaction.cpp
```

The -o option identifies the name of the executable binary. The names of the two implementation files complete the command.

To run the executable binary, we enter

```
accounting
```

Visual Studio

To compile our application at the command-line on a Windows platform using the Visual Studio compiler, we enter the command (To open the Visual Studio command prompt window, we press Start > All Programs and search for the prompt in the Visual Studio Tools sub-directory.)

```
cl /Fe accounting main.cpp Transaction.cpp
```

The /Fe option identifies the name of the executable binary. The names of the two implementation files follow this option.

To run the executable, we enter

```
accounting
```

IDE Compilation

Integrated Development Environments (IDEs) are software development applications that integrate features for coding, compiling, testing and debugging source code in different languages. The IDE used in this course is Microsoft's Visual Studio.

Build and Execute

The following steps build and execute a modular application in Visual Studio 2013 or newer:

- Start Visual Studio
- Select New Project
- Select Visual C++ -> Win32 -> Console Application
- Enter Transaction Example as the Project Name | Select OK
- Press Next
- Check Empty Project | Press Finish
- Select Project -> Add New Item
- Select Header .h file | Enter Transaction as File Name | Press OK
- Select Project -> Add New Item
- Select Implementation .cpp file | Enter Transaction as File Name | Press OK
- Select Project -> Add New Item
- Select Header .h file | Enter main as File Name | Press OK
- Select Project -> Add New Item
- Select Implementation .cpp file | Enter main as File Name | Press OK
- Select Build | Build Solution
- Select Debug | Start without Debugging

The input prompts and results of execution appear in a Visual Studio command prompt window.

Tracing

The following steps trace through execution of your program using the builtin debugger

- Select the file named main.cpp

- Move the cursor to the left-most column of the for statement in the main() function and left-click | This places a red dot in that column, which identifies a breakpoint
- Move the cursor to the left-most column of the closing brace for the function and left-click | This places a red dot in the column, which identifies another breakpoint
- Select Debug -> Start Debugging | Execution should pause at the first executable statement
- Observe the values under the Locals tab in the Window below the source code
- Press F10 and answer the three input prompts
- Select the source code Window
- Observe the values under the Locals tab in the Window below the source code
- Press F10 3 times and note the value of i
- Press F5, note where execution pauses and observe the value of i
- Press F5 again to exit

The keystrokes for the various debugging options available in this IDE are listed next to the sub-menu items under the Debug menu.

Unit Tests

Unit testing is an integral part of modular programming. A *unit test* is a code snippet that tests a single assumption in a work unit of a complete program. Each *work unit* is a single logical component with a simple interface. Typical work units are functions and classes.

We use unit tests to examine the work units in a program and rerun the tests after each upgrade. We store the test suite in a separate module.

Calculator Example

Consider a Calculator module that raises an integer to the power of an integer exponent and determines the integer exponent to which an integer base has been raised to obtain a given result. The header file for the Calculator module includes the prototypes for these two work units:

```
// Calculator.h
// ...
int power(int, int);
int exponent(int, int);
```

The suite of unit tests for this module checks if the implementations return the expected results. The header file for the Tester module contains:

```
// Tester.h

int testSuite(int BASE, int EXPONENT, int RESULT);
```

The implementation file for the Tester module contains:

```
// Tester.cpp

#include<iostream>
using namespace std;
#include "Calculator.h"

int testSuite(int BASE, int EXPONENT, int RESULT) {
    int passed = 0;
    int result;
    result = power(BASE, EXPONENT);
    if (result == RESULT) {
        cout << "Raise to Power Test Passed" << endl;
        passed++;
    }
    else {
        cout << "Raise to Power Test Failed" << endl;
    }
    result = exponent(RESULT, BASE);
    if (result == EXPONENT) {
        cout << "Find Exponent Test Passed" << endl;
        passed++;
    }
    else {
        cout << "Find Exponent Test Failed" << endl;
    }
    return passed;
}
```

A first attempt at implementing the Calculator module might look like:

```
// Calculator.cpp

#include "Calculator.h"

int power(int base, int exp) {
    int i, result = 1;
    for (i = 0; i < exp; i++)
        result *= base;
    return result;
}

int exponent(int result, int base) {
    int exp = 0;
    while(result >= base) {
        exp++;
        result /= base;
    }
    return exp;
}
```

The following test main produces the results shown on the right:

```
// Test Main
// testmain.cpp

#include<iostream>
using namespace std;
#include "Tester.h"

int main() {
    int passed = 0;
    passed += testSuite(5, 3, 125);
    passed += testSuite(5, -3, 0.008);
    cout << passed << " Tests Passed" << endl;
}
```

```
Raise to Power Test Passed
Find Exponent Test Passed
Raise to Power Test Failed
Find Exponent Test Failed
2 Tests Passed
```

The unit tests show that this implementation does not handle bases that are negative and should be upgraded.

Good Programming Practice

It is good programming practice to write the suite of unit tests for the work units in a module as soon as we have defined the header file and before starting to code the bodies of the work units. As we fill in the implementation details, we can continue testing our module to ensure that it produces the results that we expect.

1.5 Check Your progress

Q.1 What is Objective Oriented Programming?

Q.2 What are the features of Objective Oriented Programming? Explain with the help of an example.

Q.3 Write a Program in C++ for addition, multiplication, subtraction and division of two numbers.

Q.4 What is Modular Programming? Explain with the help of an example.

Block-2

1.1 Learning Objectives

After going through this unit, the learner will able to:

- Define the basic syntax
- Learn about keywords
- Define the Member Functions
- Learn about the Input and Output Example
- Learn about Dynamic memory

1.2 Basic Syntax

C++ augments the procedural features of C with object-oriented features. The notes entitled "Programming Computers Using C" describe the more common syntax that these two languages share.

This chapter introduces some basic features used in object-oriented programming, which C++ adds to C. The topics covered including types, declarations, definitions, scope, overloading and pass by reference.

Keywords

alignas	alignof	and	and_eq	asm	auto	bitand
bitor	bool	break	case	catch	char	char16_t
char32_t	class	compl	const	constexpr	const_cast	continue
decltype	default	delete	do	double	dynamic_cast	else
enum	explicit	export	extern	false	float	for
friend	goto	if	inline	int	long	mutable
namespace	new	not	not_eq	noexcept	nullptr	operator
or	or_eq	private	protected	public	register	reinterpret_cast
return	short	signed	sizeof	static	static_assert	static_cast
struct	switch	template	this	thread_local	throw	true
try	typedef	typeid	typename	union	unsigned	using
virtual	void	volatile	wchar_t	while	xor	xor_eq

A C++ compiler will successfully compile any C program that does not use any of these keywords as identifiers provided that that program satisfies C++'s type safety requirements. We call such a C program a *clean C* program.

Types

C++ supports the fundamental types of the core language and any compound types that we and other programmers define.

Fundamental Types

The fundamental types include:

- Integral Types (store data exactly in equivalent binary form)
  - bool - not in C
  - char
  - int - short, long, long long
- Floating Point Types (store data to a specified precision - can store very small and very large values)
  - float
  - double - long double

bool

bool stores a logical value: either true and false. The ! operator reverses the value: !true is false and !false is true.

! is self-inverting on bool types. (However, it is not self-inverting on other types.)

bool to int

Conversions of boolean values to integer values and vice versa require care. true promotes to an int of value 1, while false promotes to an int of value 0. Applying the ! operator to an int value other than 0 produces a value of 0, while applying the ! operator to an int value of 0 produces a value of 1. That is, the following code snippet displays 1 (not 4)

```
int x = 4;
cout << !!x;
```

```
1
```

C++ treats the integer value 0 as false and any other vaue as true.

Compound Types

A *compound type* is a type that is composed of other types. (The C language uses the term derived type.) To identify compound types that are object-oriented classes we use the keywords ***struct*** or ***class***

```
// Modular Example
// Transaction.h

struct Transaction {
    int acct;      // account number
    char type;     // credit 'c' debit 'd'
    double amount; // transaction amount
};
```

C++ does not require the keyword ***struct*** or ***class*** in prototypes or object definitions as shown on the left. The C equivalent is shown on the right.

```
// Modular Example - C++
// Transaction.h

struct Transaction {
    int acct;
    char type;
    double amount;
};
void enter(Transaction*);
void display(const Transaction*);
// ...

int main() {
    Transaction tr;
    // ...
}
```

```
// Modular Example - C
// Transaction.h

struct Transaction {
    int acct;
    char type;
    double amount;
};
void enter(struct Transaction*);
void display(const struct Transaction*);
// ...

int main() {
    struct Transaction tr;
    // ...
}
```

auto

The auto keyword deduces an object's type from its initializer. The initializer is necessary in any auto definition.

For example,

```
auto x = 4;   // x is of type int and initialized to 4
auto y = 3.5; // y is of type double and initialized to 3.5
```

auto simplifies our coding by using information that the compiler already knows. This will prove particularly useful when working with the standard libraries.

Declarations and Definitions

Declarations

A declaration associates an entity with a type. The entity may be a variable, an object or a function. That is, we use a declaration to tell the compiler how to interpret the entity's identifier.

For example, by writing the function prototype

```
int add(int, int);
```

we declare add() to be a function that receives two ints and returns an int. We do not specifying its meaning.

For example, by writing

```
struct Transaction;
```

we declare ***Transaction*** to be a class. This is called a *forward declaration*. A forward decalaration is like a function prototype: it tells the compiler that the entitiy is a valid type, but does not tell the compiler the entity's meaning.

A declaration does not have to specify a meaning, but may specify one.

Definitions

A definition is a declaration that associates a meaning with an identifier. A definition may only appear once within its code block or translation unit. This is called the *one-definition rule*.

For example, the following two definitions attach meanings to ***Transaction*** and to ***display():***

```
struct Transaction {
    int acct;      // account number
    char type;     // credit 'c' debit 'd'
    double amount; // transaction amount
};
```

```
void display(const Transaction* tr) { // definition of display

    cout << "Account " << tr->acct << endl;
    cout << (tr->type == 'd' ? " Debit $" : " Credit $") << endl;
    cout << tr->amount << endl;
}
```

We cannot redefine Transaction or display() within the same code block or translation unit.

Definitions are Executable Statements

Each definition is an executable statement. We may embed it amongst other executable statements.

For example, we may place a definition within an initializer:

```
for (int i = 0; i < n; i++)
  //...
```

Declarations are not necessarily Definitions

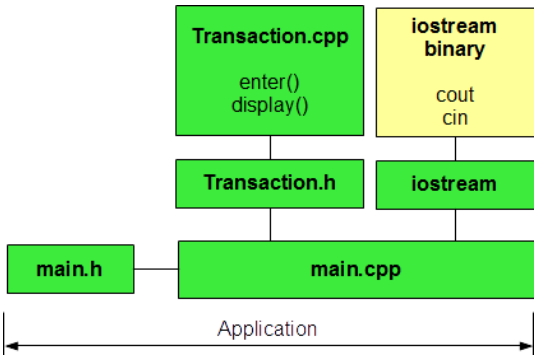
Forward declarations and function prototypes are declarations that are not definitions. They associate an identifier with a type, but do not attach any meaning to the identifier. We may repeat such declarations several times within the same code block or translation unit.

Repeated declarations may occur when we include several header files in an implementation file. However, if any header file contains a definition, the translation unit that includes that header file must not break the one-definition rule.

Designing Away Multiple Definitions

A definition that appears more than once within the same translation unit violates the one-definition rule and generates a compiler error. Consider the options shown below.

Our sample program consists of three modules: ***main***, ***Transaction*** and ***iostream***.



The ***main*** module's implementation file calls ***add(),*** which receives the address of a ***Transaction*** object:

```
// Potential Definition Conflict
// main.cpp

#include <iostream>
#include "main.h"        // prototype for add()
#include "Transaction.h" // prototypes for enter() and display()
using namespace std;

int main() {
    int i;
    double balance = 0.0;
    Transaction tr;

    for (i = 0; i < NO_TRANSACTIONS; i++) {
        enter(&tr);
        display(&tr);
        add(&balance, &tr);
    }
    cout << "Balance " << balance << endl;
}

void add(double* bal, const Transaction* tr) {
    bal += (tr->type == 'd' ? -tr->amount : tr->amount);
}
```

The Transaction module's header file defines the Transaction type:

```
// Modular Example
// Transaction.h

struct Transaction {
    int acct;      // account number
    char type;     // credit 'c' debit 'd'
    double amount; // transaction amount
};

void enter(Transaction* tr);
void display(const Transaction* tr);
```

If we place the prototype for the add() function in the main module's header file, main.cpp will not compile:

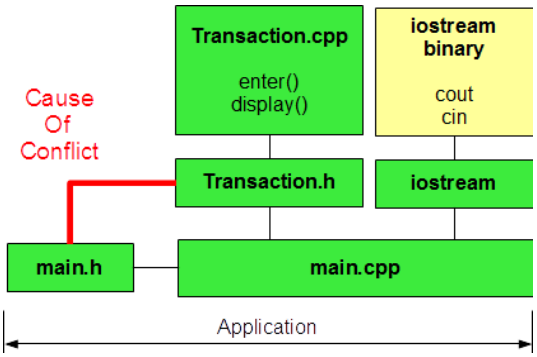
```
// main.h

#define NO_TRANSACTIONS 3

void add(double*, const Transaction*);
```

The compiler will report ***Transaction**** as undeclared. Since the compiler analyzes code sequentially, it will not know what ***Transaction*** when it encounters the prototype for ***add(),***

Including the ***Transaction.h*** in ***main.h*** would resolve this error but would break the one-definition rule:

```
// main.h

#define NO_TRANSACTIONS 3
#include "Transaction.h"  // BREAKS THE ONE-DEFINITION RULE!

void add(double*, const Transaction*);
```

The ***main.cpp*** translation unit will contain TWO definitions of ***Transaction.***



There are two possible solutions to this problem.

Forward Declaration Solution

A forward declaration informs the compiler that the identifier Transaction is a valid type, without defining the type.

```
// main.h

#define NO_TRANSACTIONS 3

struct Transaction; // forward declaration
void add(double*, const Transaction*);
```

This solution provides the compiler with just enough information to accept the identifer as a valid type.

Compact Solution
The alternative is to move the prototype from **main.h** to the **Transaction.h**:

```
// Modular Example
// Transaction.h

struct Transaction {
    int acct;      // account number
    char type;     // credit 'c' debit 'd'
    double amount; // transaction amount
};

void enter(Transaction* tr);
void display(const Transaction* tr);
void add(double*, const Transaction*);
```

This compact solution localizes all declarations related to the ***Transaction*** type within the same file. We call functions that support a class *helper functions* for that class.

Proper Order of Header File Inclusion

To avoid conflicts with system header files, we include header files in the following order:

- #include < ... > - system header files
- #include " ... " - other system header files
- #include " ... " - your own header files

We insert namespace declarations and directives after all header file inclusions.

Scope

The *scope* of a declaration is the portion of a program over which the identifier is visible.
A declaration that is visible to the entire program has *global* scope. A declaration in a code block is local to its block. We say that the declaration has *block scope*. Its scope begins at its declaration and ends at the end of its block. We say that the identifier is a *local* variable or object. It is local to the block.
A declaration with a narrower scope can shadow a declaration with a broader scope, making the latter temporarily invisible. For example, in the following program the second declaration shadows the first making the scope of the first declaration of i discontinuous:

```
// scope.cpp

#include <iostream>
using namespace std;

int main() {
    int i = 6;
    cout << i << endl;
    for (int j = 0; j < 3; j++) {
        int i = j * j;
        cout << i << endl;
    }
    cout << i << endl;
}
```

```
6
0
1
4

6
```

Going Out of Scope

When a declaration goes out of scope the program loses access to the variable or object. Identifying the precise point at which a variable's or object's declaration goes out of scope is important in object-oriented programming.

In the following code snippet, the counter i, declared within the for statement, goes out of scope immediately *after* the closing brace:

```
for (int i = 0; i < 4; i++) {
    cout << "The value of i is " << i << endl;
} // i goes out of scope here
```

We cannot refer to i after the closing brace.

A variable or object declared *within* a block goes out of scope immediately *before* the block's closing brace.

```
for (int i = 0; i < 3; i++) {
    int j = 2 * i;
    cout << "The value of j is " << j << endl;
} // j goes out of scope here
```

The scope of j extends from its definition to just before the end of each iteration. The scope of i extends throughout the iteration.

Function Rules

Function rules are slightly stricter rules in C++ than in C. These tighter rules enable language features such as overloading and pass by reference, which are absent in the C language.

Prototypes

The declaration of a function prototype consists of the function's return type, its identifier and all of its parameter types. The order of its parameter types matters. The parameter identifiers are optional.

A declaration with no parameters identifies an empty parameter list. The keyword void, which the C language uses with prototypes that have no parameters is redundant in C++. We omit the keyword in in C++.

Prototypes Required

C++ enforces type safety by requiring a declaration of the prototype for a function wherever source code calls the function before its definition. The compiler uses the prototype to check the argument types in the call against the parameter types in the prototype.

For example, the compiler will generate an error for the following program (***printf*** is undeclared):

```
int main() {
    printf("Hello C++\n");
}
```

For type safety, we include the prototype:

```
#include <cstdio>
using namespace std;

int main() {
    printf("Hello C++\n");
}
```

Overloading

In object-oriented programming, a function can have several meanings. If a function identifier has more than one definition, we say that that function is *overloaded*.

C++ distinguishes function definitions by their identifier, their parameter types and the order of their parameter types. Two functions with the same identifier but different parameter types or differently ordered parameter types are distinct functions.

For example, to display data on the standard output device, we can define a display() function with different parameter types:

```
void display(int x) {
    cout << x << endl;
}

void display(int* x, int n) {
    for (int i = 0; i < n; i++)
        cout << x[i] << ' ';
    cout << endl;
}
```

The compiler generates two separate definitions of ***display():*** one for each set of parameters. The linker binds each function call to the appropriate definition based on the argument types in the function call.

Signature

A function's *signature* identifies the function uniquely and consists of

- the identifier
- the parameter types (without any const modifiers or address of operators)
- the order of the parameter types

```
type identifier ( type identifier [, ... , type identifier] )
```

The brackets enclose optional information. The return type and the parameter identifiers are not part of a function's signature.

The compiler preserves uniqueness by renaming each function using a combination of its identifier, its parameter types and the order of its parameter types. We refer to this renaming as *mangling*.

Default Parameter Values

We can assign default values to some or all of a function's parameters. We must however arrange the parameters with the default values as the rightmost parameters. We specify the default values in the first function declaration in a translation unit.

A declaration with default parameter values takes the following form:

```
type identifier(type[, ...], type = value);
```

The assignment operator followed by a value identifies the default value for each paramter.

Specifying default values for function parameters reduces the need for coding multiple function definitions where the function logic is identical in every respect except for the values of the parameters.

For example,

```
// Default Parameter Values
// default.cpp

#include <iostream>
using namespace std;

void display(int, int = 5, int = 0);

int main() {

    display(6, 7, 8);
    display(6);
    display(3, 4);
}

void display(int a, int b, int c) {
    cout << a << ", " << b << ", " << c << '\n';
}
```

```
6, 7, 8
6, 5, 0
3, 4, 0
```

Each call to ***display()*** must include enough arguments to initialize the parameters that don't have default values. In this example, each call must include at least one argument. An argument corresponding to a parameter that has a default value override the default value.

Pass By Reference

Pass by reference is an alternative mechanism to passing by address available in C++. Pass-by-reference code is notably more readable than pass-by-address code.

The declaration of a function parameter that is passed by reference takes the form

```
type identifier(type& identifier, ... )
```

The **&** identifies the parameter as an alias for, rather than a copy of, the corresponding argument in the function call. The identifier is the alias name within the function definition. Any change to the value of a parameter received by reference changes the value of the corresponding argument in the function call.

Comparison Example

Consider a function that swaps the values stored in two different memory locations. The programs listed below compare pass-by-address and pass-by-reference solutions. The program on the left passes by address using pointers. The program on the right passes by reference:

```
// Swapping values by address
// swap1.cpp

#include <iostream>
using namespace std;
void swap ( char *a, char *b );

int main ( ) {
    char left;
    char right;

    cout << "left  is ";
    cin  >> left;
    cout << "right is ";
    cin  >> right;

    swap(&left, &right);

    cout << "After swap:"
           "\nleft  is " <<
           left <<
           "\nright is " <<
           right <<
           endl;
}

void swap ( char *a, char *b ) {
    char c;

    c = *a;
    *a = *b;
    *b = c;
}
```

```
// Swapping values by reference
// swap2.cpp

#include <iostream>
using namespace std;
void swap ( char &a, char &b );

int main ( ) {
    char left;
    char right;

    cout << "left  is ";
    cin  >> left;
    cout << "right is ";
    cin  >> right;

    swap(left, right);

    cout << "After swap:"
           "\nleft  is " <<
           left <<
           "\nright is " <<
           right <<
           endl;
}

void swap ( char &a, char &b ) {
    char c;

    c = a;
    a = b;
    b = c;
}
```

Reference syntax is slightly simpler. To pass an object by reference, we attach the address of operator to the parameter type. This operator instructs the compiler to pass by reference. The corresponding arguments in the function call and the object names within the function definition are not prefixed by the dereferencing operator required in passing by address.

Technically, the compiler converts each reference to a pointer with an unmodifiable address.

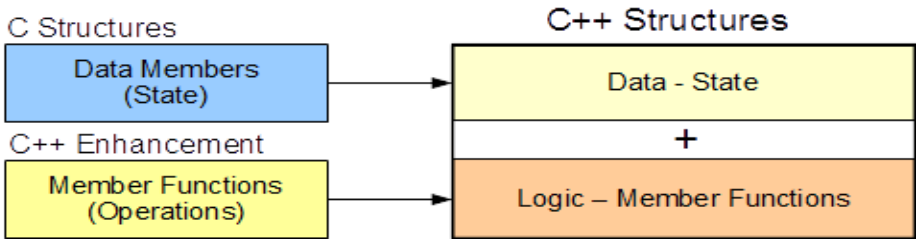
1.3 Member Functions and Privacy

Encapsulation incorporates within a class the structure of data that its objects hold with the logic that operates on that data to create a clean interface between the class and its clients and hide the implementation details from them. In C++, we incorporate logic through member functions. The data members of a class hold information about the structure its objects' state while the member functions define operations that query, modify and manage that state.

This chapter describes the C++ syntax for declaring member functions in the definition of a class, defining the member functions in the implementation file and limiting accessibility to the data values of an object.

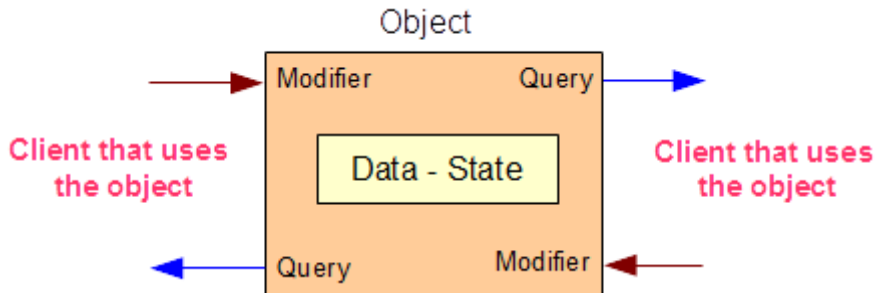
Member Functions

Member functions provide communication links between a client and an object. The client calls the object's member functions to access the object's data and possibly to change its data.



We can classify member functions under three mutually exclusive categories:

- *queries* - accessor methods - report the state of the object
- *modifiers* - mutator methods - change the state of the object
- *special* - manager methods - create, assign and destroy an object



Every member function has direct access to the data members of its class as well as its other member functions. Each member function receives and passes information between the client and its object through parameters and a return value.

Consider a Student type with the following definition

```
struct Student {
    int no;
    char grade[14];
};
```

Adding a Member Function

Declaration

To declare a member function, we insert its prototype into the definition of its class.
For example, to add ***display()*** as a member to our ***Student*** type, we write:

```
struct Student {
    int no;
    char grade[14];

    void display() const; // member function
};
```

The ***const*** qualifier identifies the member function as a query. A query cannot change the state of its object. That is, this query cannot change the value of ***no*** or any character in ***grade.***

As a member function, ***display()*** has direct access to the object's variables (***no*** and ***grade***). There is no need to pass these values as parameters in the function prototype.

Definition

We define ***display()*** in the implementation file as follows:

```
void Student::display() const {

    cout << no << ' ' << grade;

}
```

The definition consists of four elements:

- the ***Student::*** prefix on the function name identifies it as a member of our ***Student*** type
- the empty parameter list - this function does not receive any values from the client or pass any values through the parameter list to the client
- the ***const*** qualifier identifies this function as a query - this function cannot change any of the values of the data members
- the data members - the function accesses ***no*** and ***grade***, which are defined within the class but outside the function

Call a Member Function

The client calls a member function in the same way that an instance of ***a struct refers to one of its data members. The call consists of the object's identifier, the . operator and the member function's identifier.***

For example, if ***harry*** is a ***Student*** object, we display its data by calling ***display()*** on ***harry***:

```
Student harry = {975, {'A','B','C'}};
harry.display();
cout << endl;
```

The object part of the function call (the part before the member selection operator) identifies the data that the function should access.

Scope of a Member Function

The scope of a member function is the scope of its class. That is, a member function can access any other member within class scope. For example, a member function can access another member function directly:

```
void Student::displayNo() const {
    std::cout << no;
}

void Student::display() const {
    displayNo(); // calls the member function defined above
    std::cout << ' ' << grade;
}
```

Accessing Global Functions

A member function can also access a function outside its class' scope. Let us add the following global function definition to the above definitions:

```
void displayNo() {
    std::cout << "Number...\n";
}
```

This global function shares the same identifier with one of the member functions. This definition does not introduce a conflict, since the client calls each function using different syntax.

```
displayNo();       // calls the global display function
harry.displayNo(); // calls the member function on harry
```

To access the global function from within the member function we apply the scope resolution operator:

```
void Student::display() const {
    ::displayNo(); // calls the global function
    displayNo();   // calls the member function
    std::cout << ' ' << grade;
}
```

Privacy

Data privacy is important in obect-oriented programming. Data members defined using the struct keyword are exposed to any client. To limit accessibility to any member, C++ lets us hide member information by classify that member as private.

In an object-oriented solution, the only members that a client should need to access are the class's communication links. The client should not need direct access to the data that describes an object's state.

Accessibility Labels

To prohibit external access to any member (data or function), we insert the label private into the definition of our class:

```
private:
```

private identifies all subsequent members listed in the class definition as inaccessible.

To allow client access, we insert the label public:

```
public:
```

public identifies all subsequent members listed in the class definition as accessible.

For example, in order to

- hide the data members of each Student object
- expose the member function(s) of the Student type

we insert the accessibility keywords as follows

```
struct Student {
  private:
    int no;
    char grade[14];
  public:
    void display() const;
};
```

The keyword ***struct*** identifies a class that is *public by default*.

class

The keyword class identifies a class that is *private by default*.

We use the keyword class to simplify the definition of a Student type

```
class Student {
    int no;
    char grade[14];
  public:
    void display() const;
};
```

The class keyword is much more common in object-oriented programming than the ***struct*** keyword. (The C language does not support privacy and a derived type in C can only take the form of a ***struct***).

Any attempt to access a member that is private generates a complier error:

```
void foo(const Student& harry) {

    cout << harry.no;  // ERROR - member is private!

}
```

The function ***foo()*** can only access the data stored in ***harry*** indirectly through ***public*** member function ***display().***

```
void foo(const Student& harry) {

    harry.display(); // OK

}
```

Modifying Private Data

If the data members of a class are private, clients cannot initialize their values directly. We need a separate member function for this specific task.

For example, to store data in Student objects, let us introduce a public modifier named set():

```
class Student {
    int no;
    char grade[14];
  public:
    void set(int n, const char* g);
    void display() const;
};
```

set() receives a student number and the address of a C-style string that contains the grades from a client and stores this information in the object's data members:

```
void Student::set(int n, const char* g) {
    int i;

    no = n; // store the Student number as received

    // store the grades as received within the available space
    for (i = 0; g[i] != '\0' && i < 13; i++) {
        grade[i] = g[i];
    grade[i] = '\0';        // set the last byte to null
}
```

Communications Links

The ***set()*** and ***display()*** member functions are the only communication links to any client. Clients can call ***set()*** or ***display()*** on any Student object, but none can access the data stored within any Student object directly.

For example, the compiler traps the following privacy breach:

```
Student harry;

harry.set(975, "ABC");
harry.display();

cout << harry.no; // ERROR .no IS PRIVATE!
```

Empty State

Hiding data members from clients gives us control over which data to accept, which to reject and which data to expose to clients. Before storing any values received from a client we can validate the incoming information. If the data is invalid, we reject it and store default values that identify the object's state as *empty*.

Upgrade *set()*

Let us upgrade our set() member function to validate incoming data; that is, to accept incoming data *only if* the student number is positive-valued and the grades are A, B, C, D or F, without exception. If any incoming data fails to meet all of these conditions, let us ignore all of the incoming data and store values that identify an *empty state:*

```
void Student::set(int n, const char* g) {
    int i;
    bool valid = true; // assume valid input, check invalidity

    if (n < 1)
        valid = false;
    else {
        for (i = 0; g[i] != '\0' && valid; i++)
            valid = g[i] >= 'A' && g[i] <= 'F' && g[i] != 'E';

    if (valid) {
        // accept client's data
        no = n;
        for (i = 0; g[i] != '\0' && i < 13; i++)
            grade[i] = g[i];
        grade[i] = '\0';  // set the last byte to the null byte
    }
    else {
        // ignore client's data, set an empty state
        no = 0;
        grade[0] = '\0';
    }
}
```

This validation logic ensures that the data stored in a Student object is either valid data or data that identifies an *empty state*.

Design Tip

Select one data member to hold a special value that identifies an empty state. Then, to determine if an object is in an empty state, all we need to do is interrogate that data member.

Upgrading *display()*

To improve this upgrade, we ensure that our display() member function executes gracefully if our object happens to be in an empty state:

```
void Student::display() const {
    if (no != 0)
        cout << no << ' ' << grade;
    else
        cout << "no data available";
}
```

Completing the Upgrade

This upgrade still leaves the data in a Student object undefined before the first call to set(). To address this deficiency, we will introduce a special member function in the chapter entitled Classes.

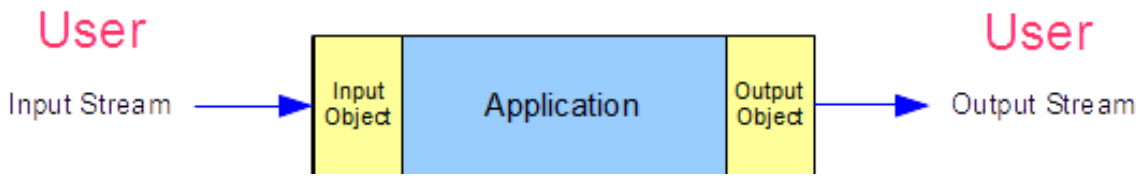
1.4 Input and Output Examples

The input and output objects introduced in the first chapter include public member functions for formatting data passing through the objects. These objects and the classes that define their structure belong to the ***iostream*** library module and are defined in its ***<iostream>*** header file. The public member functions report the state of each object as well as control the formatting.

This chapter describes the input and output objects in detail along with their member functions. This provides examples of the role that member functions play and sets the stage for subsequent coding of member functions for our custom classes.

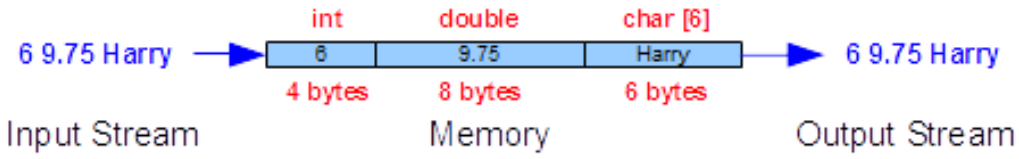
Streams

Data enters an application in one stream and leaves the application in another stream. A *stream* is a sequence of characters without limitation. The number of characters in a stream can be indeterminate. An input object stroes data from an input stream in the application's memory. An output object copies data from the application's memory into an output stream. Input and output objects operate in FIFO (First In First Out) order. The first character entering the input object is the first character stored in memory.



The standard input and output objects of the ***iostream*** library represent the standard peripheral devices, such as the keyboard and display.

The input object converts a sequence of characters from its attached input stream into values of a specified type stored in system memory. The output object converts values of a specified type stored in system memory into a sequence of characters in its associated output stream. These objects use the data type associated with the region of memory that holds each data value to make the appropriate conversions from or to the sequence of characters.



The data in a stream, unlike the data stored in a region of memory, is not associated with any type. The notion of type is system memory specific.

Output Objects

An output object is an instance of the ostream type, which defines the structure of an output device. An ostream object copies data from system memory into an output stream; in copying, it converts the data in system memory into a sequence of characters.
The iostream module defines three standard output objects:

- cout - transfers a buffered sequence of characters to the standard output device
- cerr - transfers an unbuffered sequence of characters to the standard error output device
- clog - transfers a buffered sequence of characters to the standard error output device

Inserting Data

The expression for inserting data into an output stream takes the form

```
output << identifier
```

where ***output*** is the name of the ***ostream*** object. << is the insertion operator. ***identifier*** is the name of the variable or object that holds the data.

For example,

```
int i = 6;
char c = ' ';
double x = 9.75;
char s[] = "Harry";
cout << i;
cout << c;
cout << x;
cout << c;
cout << s;
cout << endl;
cerr << "Data has been written";
```

```
6 9.75 Harry

Data has been written
```

Each expression with an ostream object as its left operand converts the data in its right operand into a sequence of characters based on the right operand's type.

endl inserts a newline character into the stream and flushes the stream's buffer.

Cascaded Insertion

We may combine these expressions into a single statement that specifies multiple insertions:

```
int i = 6;
char c = ' ';
double x = 9.75;
char s[] = "Harry";

cout << i << c << x << c << s << endl;

cerr << "Data has been written";
```

```
6 9.75 Harry
Data has been written
```

We call such repeated use of the insertion operator ***cascading***.

Member Functions

The ostream type supports the following public member functions for formatting conversions:

- width(int) - sets the field width to the integer received
- fill(char) - sets the padding character to the character received
- setf(...) - sets a formatting flag to the flag received
- unsetf(...) - unsets a formatting flag for the flag received
- precision(int) - sets the decimal precision to the integer received

width

The ***width(int)*** member function specifies the minimum width of the next output field:

```
// Field Width
// width.cpp

#include <iostream>
using namespace std;

int main() {
    int attendance = 27;
    cout << "1234567890" << endl;
    cout.width(10);
    cout << attendance << endl;
    cout << attendance << endl;
}
```

```
1234567890
        27
27
```

width(int) applies only to the next field. Note how the field width for the first display of attendance is 10, while the field width for the second display of attendance is just the minimum number of characters needed to display the value (2).

fill

The ***fill(char)*** member function defines the padding character. The output object inserts this character into the stream wherever text occupies less space than the specified field width. The default fill character is ' ' (space). To pad a field with ***'*'s***, we add:

```
// Padding
// fill.cpp

#include <iostream>
using namespace std;

int main() {
    int attendance = 27;
    cout << "1234567890" << endl;
    cout.fill('*');
    cout.width(10);
    cout << attendance << endl;
}
```

```
1234567890

********27
```

The padding character remains unchanged, until we reset it.

setf, unsetf

The **setf()** and **unsetf()** member functions control formatting and alignment. Their control flags include:

Control Flag	Result
ios::fixed	ddd.ddd
ios::scientific	d.dddddddEdd
ios::left	align left
ios::right	align right

The scope resolution (ios::) on these flags identifies them as part of the ios class.

setf, unsetf – Formatting

The default format in C++ is *general format*, which outputs data in the simplest, most succinct way possible (1.34, 1.345E10, 1.345E-20). To output a fixed number of decimal places, we select *fixed format*. To specify fixed format, we pass ***the ios::fixed*** flag to ***setf():***

```
// Fixed Format
// fixed.cpp

#include <iostream>
using namespace std;

int main() {
    double pi = 3.141592653;
    cout << "1234567890" << endl;
    cout.width(10);
    cout.setf(ios::fixed);
    cout << pi << endl;
}
```

```
1234567890
  3.141593
```

Format settings persist until we change them. To unset fixed format, we pass the ios::fixed flag to the unsetf() member function:

```
// Unset Fixed Format
// unsetf.cpp

#include <iostream>
using namespace std;

int main() {
    double pi = 3.141592653;
    cout << "1234567890" << endl;
    cout.width(10);
    cout.setf(ios::fixed);
    cout << pi << endl;
    cout.unsetf(ios::fixed);
    cout << pi << endl;
}
```

```
1234567890
  3.141593
3.14159
```

To specify scientific format, we pass the ios::scientific flag to the setf() member function:

```
// Scientific Format
// scientific.cpp

#include <iostream>
using namespace std;

int main() {
    double pi = 3.141592653;
    cout << "12345678901234" << endl;
    cout.width(14);
    cout.setf(ios::scientific);
    cout << pi << endl;
}
```

```
12345678901234
  3.141593e+00
```

To turn off scientific format, we pass the ***ios::scientific*** flag to the ***unsetf()*** member function.

setf, unsetf – Alignment

The default alignment is right-justified.

To specify left-justification, we pass the ios::left flag to the setf() member function:

```
// Left Justified
// left.cpp

#include <iostream>
using namespace std;

int main() {
    double pi = 3.141592653;
    cout << "1234567890" << endl;
    cout.width(10);
    cout.fill('?');
    cout.setf(ios::left);
    cout << pi << endl;
}
```

```
1234567890
3.14159???
```

To turn off left-justification, we pass the ios::left flag to the unsetf() member function:

```
cout.unsetf(ios::left);
```

precision

The ***precision()*** member function sets the precision of subsequent floating-point fields. The default precision is ***6*** units. General, fixed, and scientific formats implement precision differently. General format counts the number of significant digits. Scientific and fixed formats count the number of digits following the decimal point.

For a precision of 2 under general format, we write

```
// Precision
// precison.cpp

#include <iostream>
using namespace std;

int main() {
    double pi = 3.141592653;
    cout << "1234567890" << endl;
    cout.setf(ios::fixed);
    cout.width(10);
    cout.precision(2);
    cout << pi << endl;
}
```

```
1234567890
      3.14
```

The precision setting applies to the output of all subsequent floating-point values until we change it.

Manipulators

Manipulators are the elegant alternative to member function calls. Manipulators are operands to the insertion operator. Manipulators that don't take arguments do not include parentheses and are defined in **<*iostream*>.** Those that take arguments include parentheses and are defined in **<*iomanip*>.** We must include **<*iomanip*>** whenever we use manipulators that take arguments.

The insertion manipulators include:

Manipulator	Effect
`fixed`	output floating-point numbers in fixed-point format
`scientific`	output floating-point numbers in scientific format
`left`	left justify
`right`	right justify
`endl`	output end of line and flush the buffer
`setprecision(int)`	set the precision of the output
`setfill(int)`	set the fill character for the field width
`setw(int)`	set the field width for the next output operand only
`setbase(int)`	set the base of the number system for `int` output
`flush`	flush the output buffer

Manipulators (except for setw(i) which only modifies the format setting for the next object) modify the format settings until we change them.

For example,

```
cout << fixed << left << setw(5) <<
        setprecision(1) << 12.376 <<
        setprecision(5) << 12.376 <<
        endl;
```

```
12.4 12.37600
```

Reference Example

The following program produces the output listed on the right

```
#include <iostream>
#include <iomanip>
using namespace std;

int main( ) {
    /* integers */
    cout << "\n* ints *\n"
    << "1234567890\n"
    << "----------\n"
    << 4321 << '\n'
    << setw(7) << 4321 << '\n'
    << setw(7) << setfill('0') << 4321 << setfill(' ')<<'\n'
    << setw(7) << left << 4321 << right << '\n';
    /* floats */
    cout << "\n* floats *\n"
    << "1234567890\n"
    << "----------\n"
    << 4321.9876546F << '\n';
    /* doubles */
    cout << "\n* doubles *\n"
    << "1234567890\n"
    << "----------\n"
    << fixed << 4.9876546 << '\n'
    << setw(7) << setprecision(3) << 4.9876546 << '\n'
    << setw(7) << setfill('0') << 4.9876546 << '\n'
    << setw(7) << left << 4.9876546 << right << '\n';
    /* characters */
    cout << "\n* chars *\n"
    << "1234567890\n"
    << "----------\n"
    << 'd' << '\n'
    << int('d') << '\n';
}
```

```
* ints *
1234567890
----------
4321
   4321
0004321
4321

* floats *
1234567890
----------
4321.99

* doubles *
1234567890
----------
4.987655
  4.988
004.988
4.98800

* chars *
1234567890
----------
d
100
```

Notes:

- a double or a float rounds to the requested precision
- char data displays in either character or decimal format

to output its numeric code, we cast the value to an int (the value output for 'd' here is its ASCII value).

Input Object

The input object is an instance of the ***istream*** type, which defines the structure of an input device. The object extracts data from the input stream and stores it in system memory, converting the sequence of characters in the stream into equivalent values in system memory.

Extraction

The expression for extracting characters from an input stream takes the form

```
inputObject >> identifier
```

where *inputObject* is the name of the input object. >> is the extraction operator. *identifier* is the name of the destination variable.

The ***iostream*** library defines one standard input object for buffered input: ***cin***.

For example,

```
int i;
char c;
double x;
char s[8];
cout << "Enter an integer,\n"
        "a character,\n"
        "a floating-point number and\n"
        "a string : " << flush;
cin >> i;
cin >> c;
cin >> x;
cin >> s;  // possible overflow
cout << "Entered " << i << ' '
     << c << ' ' << x << ' ' << s << endl;
```

```
Enter an integer,
a character,
a floating-point and
a string : 6  -  9.75  Harry

Entered 6 - 9.75 Harry
```

Each expression with an istream object as its left operand converts the next sequence of characters into a value stored in the data type of its right operand.

The cin object skips leading whitespace with numeric, string and character types (in the same way that scanf("%d"...), scanf("%lf"...), scanf("%s"...) and scanf(" %c"...) skip whitespace in C).

```
// Leading Whitespace
// leading.cpp

#include <iostream>
using namespace std;

int main() {
    char str[11];

    cout << "Enter a string : " << endl;
    cin >> str;
    cout << "|" << str << "|" << endl;
}
```

```
        Note: _ denotes space



Enter a string :
__abc
|abc|
```

Whitespace

cin treats whitespace in the input stream as a delimiter for numeric and string data types. For C-style null-terminated string types, cin adds the null byte after the last non-whitespace character stored in memory:

```
// Trailing Whitespace
// trailing.cpp

#include <iostream>
using namespace std;

int main() {
    char str[11];

    cout << "Enter a string : " << endl;
    cin >> str;
    cout << "|" << str << "|" << endl;
}
```

```
        Note: _ denotes space



Enter a string :
__abc__
|abc|
```

Cascaded Extraction

The extraction operator (>>), like the insertion operator, processes cascaded input:

```
int i;
char c;
double x;
char s[8];
cout << "Enter an integer,\n"
        "a character,\n"
        "a floating-point number and\n"
        "a string : " << flush;
cin >> i >> c >> x >> s;
cout << "Entered " << i << ' '
     << c << ' ' << x << ' ' << s << endl;
```

```
Enter an integer,
a character,
a floating-point and
a string : 6  -  9.75  Harry

Entered 6 - 9.75 Harry
```

Note that reading input in this manner is discouraged (see below).

Overflow

In the above two examples overflow is possible while filling s, since the extraction operator >> does not restrict the number of characters accepted. If more than 7 characters are in the input stream some of the data stored may corrupt other memory as shown on the right:

```
// Overflow
// overflow.cpp

#include <iostream>
using namespace std;

int main() {
    int i;
    char c;
    double x;
    char s[8];
    cout << "Enter an integer,\n"
        "a character,\n"
        "a floating-point number and\n"
        "a string : \n";
    cin >> i >> c >> x >> s;
    cout << "Entered " << endl;
    cout << i << ' '
         << c << ' ' << x << ' ' << s << endl;
}
```

```
Enter an integer,
a character,
a floating-point and
a string :
6 - 9.75 Constantinople

Entered
6 - 2.04952 Constantinople
```

The corrupted result varies from platform to platform.

Member Functions

The istream type supports the following member functions:

- ignore(...) - ignores/discards character(s) from the input buffer
- get(...) - extracts a character or a string from the input buffer
- getline(...) - extracts a line of characters from the input buffer

ignore

The ignore() member function extracts characters from the input buffer and discards them. ignore() doesn't skip leading whitespace. Two versions of ignore() are available:

```
cin.ignore();
cin.ignore(2000, '\n');
```

The no-argument version discards a single character. The two-argument version removes and discards up to the specified number of characters or up to the specified delimiting character, whichever occurs first and discards the delimiting character. The default delimiter is end-of-file (not end-of-line).

get

The ***get()*** member function extracts either a single character or a string from the input buffer. Three versions are available:

```
// Input Extraction Using get()
// get.cpp

#include <iostream>
using namespace std;

int main() {
    char c, d, t[8], u[8], v;

    c = cin.get();          // extracts a single character
    cin.get(d);             // extracts a single character
    cin.get(t, 8);          // newline delimiter - accepts up to 7 chars
                            //    and adds a null byte
    cin.ignore(2000, '\n'); // extracts the 'j' and the newline
    cin.get(u, 8, '\t');    // tab delimiter - accepts up to 7 chars and
                            //    adds a null byte
    cin.ignore();           // extracts the tab
    cin.get(v);             // extracts a single character

    cout << "c = " << c << endl;
    cout << "d = " << d << endl;
    cout << "t = " << t << endl;
    cout << "u = " << u << endl;
    cout << "v = " << v << endl;
}
```

The above program produces the following results:

```
Input stream : abcdefghij
               klmn\topqr
Output:
-------
c = a
d = b
t = cdefghi
u = klmn
v = o
```

get() does not skip leading whitespace. ***get(,)*** leaves the delimiting character in the input buffer. If we use ***get(,)*** we need to remove the delimiting character, if there is one. Both string versions - ***get(char*, int)*** and ***get(char*, int, char)*** - append a null byte to the sequence of characters stored in memory.

getline

getline() behaves like ***get(),*** but extracts the delimiter from the input buffer:

```
// Input Extraction Using getline()
// getline.cpp

#include <iostream>
using namespace std;

int main() {
    char t[8], u[8], v;

    cin.getline(t, 8);        // newline delimiter - accepts up to 7 chars
                              //    and adds a null byte
    cin.getline(u, 8, '\t'); // tab delimiter - accepts up to 7 chars and
                              //    adds a null byte
    cin.get(v);               // extracts a single character

    cout << "t = " << t << endl;
    cout << "u = " << u << endl;
    cout << "v = " << v << endl;
}
```

The above program produces the following results:

```
Input stream : cdefghi
               jklmn\topqr
Output:
-------
t = cdefghi
u = jklmn
v = o
```

getline(), like ***get(),*** does not skip leading whitespace and appends a null byte to the sequence of characters stored in system memory.

Format Control

Manipulators

The manipulators for input objects are listed below:

Manipulator	Effect
`skipws`	skip whitespace
`noskipws`	turn off skip whitespace
`setw(int)`	set the field width for next input (strings only)

The argument to ***setw()*** should be one more than the maximum number of input characters to be read. Note that the ***setw()*** manipulator is an alternative to ***get(char*, int),*** but ***setw()*** skips leading whitespace unless we turn off skipping.

Once a manipulator has modified the format settings of an input object, those settings remain modified.

We may combine manipulators with input variables directly. For example,

```
// Input Manipulators
// manipulator.cpp

#include <iostream>
#include <iomanip>
using namespace std;

int main( ) {
    char a[5], b[2], c, d[7];
    cout << "Enter : ";
    cin >> setw(5) >> a >>
           setw(2) >> b >> noskipws >>
           c >> skipws >> d;
    cout << "Stored '" << a <<
            "' & '" <<  b <<
            "' & '" << c <<
            "' & '" << d << "'" << endl;
}
```

```
Enter :     abcde          fgh


Stored 'abcd' & 'e' & '' & 'fgh'
```

State

The ***ostream*** and ***istream*** types expose member functions for reporting and changing the state of their objects. These functions include:

- ***good()*** - the next operation might succeed
- ***fail()*** - the next operation will fail
- ***eof()*** - end of data has been encountered
- ***bad()*** - the data may be corrupted
- ***clear()*** - reset the state to good

We should check the state of the input object every time we extract a sequence of characters from the input buffer. If the object has encountered an invalid character, the object will fail and leave the invalid character in the input buffer. The fail() function will return true.

Before a failed object can continue extracting data, we must clear it of its failed state. The clear() function resets the object's state to good:

```
if(cin.fail()) {            // checks if cin is in a failed state
    cin.clear();            // clears state to allow further extraction
    cin.ignore(2000, '\n'); // clears the input buffer
}
```

See the following section for a complete example that evaluates the state of the input object.

Robust Validation

The state functions help us validate input robustly. We check the input object's state after each extraction of a sequence of characters to ensure that the object converted a value and that the converted value is within valid admissible bounds. We reject invalid input and out-of-bound values, resetting any failed state, and requesting fresh input as necessary.

getPosInt

To extract a positive int that is not greater than max from the standard input device, we write

```
// getPosInt extracts a positive integer <= max
// from standard input and returns its value
//
int getPosInt(int max) {
    int value;
    int keepreading;

    keepreading = 1;
    do {
        cout << "Enter a positive integer (<= " << max << ") : ";
        cin  >> value;

        if (cin.fail()) {   // check for invalid character
            cerr << "Invalid character.  Try Again." << endl;
            cin.clear();
            cin.ignore(2000, '\n');
        } else if (value <= 0 || value > max) {
            cerr << value << " is outside the range [1," <<
                 max << ']' << endl;
            cerr << "Invalid input.  Try Again." << endl;
            cin.ignore(2000, '\n');
        // you may choose to omit this branch
        } else if (char(cin.get()) != '\n') {
            cerr << "Trailing characters.  Try Again." << endl;
            cin.ignore(2000, '\n');
        } else
            keepreading = 0;
    } while(keepreading == 1);

    return value;
}
```

1.5 Dynamic Memory

Reusability is an important consideration in object-oriented design. We aim to reuse the software that we write in other applications with no or slight modification. Objects are more reusable by different clients if they account for their memory needs internally. Memory needs may depend on the size of a problem and in some cases may not be known even approximately until run-time. By designing objects to allocate memory at run-time, we create more flexible programming solutions.

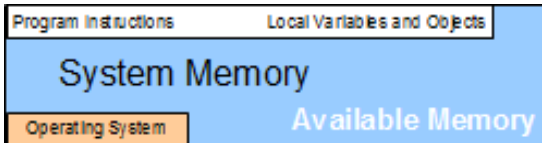
This chapter introduces the basic C++ syntax for allocating and deallocating memory dynamically in preparation for designing classes with variable memory needs. The chapter entitled Classes and Resources covers the details involved in coding classes that allocate memory at run-time.

System Memory

After an operating system loads an executable into RAM, it transfers control to the entry point of the executable (the ***main()*** function). The executable includes memory allocated at compile time. Throughout execution, the application itself may request more memory from the operating system. The system attempts to satisfy such requests by reserving more memory in RAM. After the application terminates and returns control to the operating system, the system recovers all of the memory that the application used.

Static Memory

The memory that the operating system allocates for the application at load time is called *static memory*. Static memory includes the space allocated for program instructions, local variables and local objects. The compiler determines the amount of static memory that each translation unit requires. The linker sets the amount of static memory that the entire application requires.



The application's local variables and objects share static memory amongst themselves. When a variable or object goes out of scope the space becomes available for a newly defined variable or

object. The lifetime of each local variable and object extends from its definition to the closing brace of the code block within which it is defined:

```
// lifetime of a local variable or an object

for (int i = 0; i < 10; i++) {
    double x = 0;       // lifetime of x starts here
    // ...
}                       // lifetime of x ends here

for (int i = 0; i < 10; i++) {
    double y = 4;       // lifetime of y starts here
    // ...
}                       // lifetime of y ends here
```

Note that the variable ***y*** may occupy the same physical memory location in RAM as variable ***x***. This system of sharing memory amongst local variables and objects ensures that application uses RAM as efficiently as possible.

Static memory is determined at compile-link time and cannot be changed during execution. This memory is fast, fixed in its amount and allocated at load time.

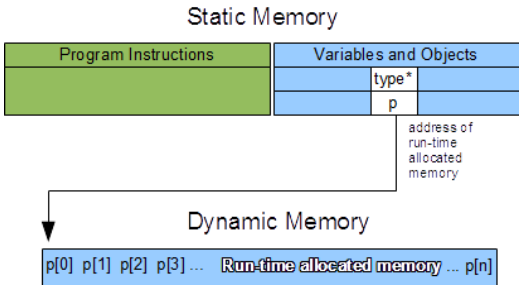
Dynamic Memory

The memory that an application requests from the operating system during execution is called *dynamic memory*.

Dynamic memory is completely distinct from the static memory that the operating system loads for the application. The operating system reserves dynamic memory at run-time and the application itself allocates and deallocates regions within this reserved memory.

To keep track of the dynamic memory currently allocated by the application, we store the address of each region in a pointer variable. We allocate memory for this pointer in static memory and must keep it alive as long as we need access to that region of dynamic memory.

Consider allocating dynamic memory for an array of n elements. We store the array's address in a pointer, ***p***, in static memory as illustrated below. We allocate the array itself dynamically and store the data in its elements sequentially starting at address ***p***.



Lifetime

The lifetime of any dynamic variable or object ends when the pointer that holds its address goes out of scope. The application must explicitly deallocate the region of dynamic memory reserved for that variable or object before this happens. If the application fails to deallocate the dynamic memory reserved for a variable or object, the memory becomes inaccessible and survives until the application reverts control back to the operating system.

Unlike variables and objects that have been allocated in static memory, those in dynamic memory do not automatically go of out scope at the closing brace of the code block within which they were defined. We must manage their deallocation explicitly ourselves.

Dynamic Allocation

The keyword ***new*** followed by [***n***] allocates contiguous space in dynamic memory for an array of n elements and returns the address of the start of that array.

A dynamic allocation statement takes the form

```
pointer = new Type[size];
```

where **Type** is the type of the array's elements.

For example, to allocate dynamic memory for an array of ***n Students***, we write

```
int n;                  // holds the number of students
Student* cpa = nullptr; // will hold the address of the dynamic array

cout << "How many students? ";
cin >> n;

cpa = new Student[n]; // allocates space in dynamic memory
```

The nullptr keyword identifies the address pointed to as the null address, an implementation constant. Initialization to nullptr ensures that cpa is not pointing to any valid address. The size of the array is a run-time variable and not an integer constant or constant expression as required for static memory allocation.

Dynamic Deallocation

The keyword ***delete*** followed by **[]** and the address of the region of dynamic memory deallocates the memory that the ***new*[]** operator had allocated.

A dynamic array deallocation takes the form

```
delete [] pointer;
```

where *pointer* holds the address of the start of the dynamically allocated array.

For example, to deallocate the memory allocated for the array of **n *Students*** above, we write

```
delete [] cpa;
cpa = nullptr;  // optional
```

The ***nullptr*** assignment ensures that ***cpa*** now holds the null address. This optional assignment eliminates the possibility of deleting the original address a second time, which is a serious run-time error. Deleting the ***nullptr*** address has no effect.

Omitting the brackets in a deallocation expression deallocates the first element of the array and leaves the other elements unreachable.

Deallocation does not return dynamic memory to the operating system. The deallocated memory remains available for subsequent re-allocations. The operating system only reclaims dynamic memory once the application reverts control back to the system.

A Complete Example

Consider a simple program in which the user enters the number of Students, the program allocates memory for that number, the user enters data for each student, the program displays the data accepted and finally the program terminates:

```
// Dynamic Memory Allocation
 // dynamic.cpp

 #include <iostream>
 #include <cstring>
 using namespace std;

 class Student {
     int no;
     char grade[14];
   public:
     void set(int, const char*);
     void display() const;
 };

 void Student::set(int n, const char* g) {
     if (n > 0) {
         no = n;
         strncpy(grade, g, 13);
     } else {
         no = 0;
         grade[0] = '\0';
     }
 }

 void Student::display() const {
     if (no != 0)
         cout << no << '' << grade;
 }

 int main( ) {
     int n;
     Student* cpa = nullptr;

     cout << "Enter the number of students : ";
     cin >> n;
     cpa = new Student[n];

     for (int i = 0; i < n; i++) {
         int no;
         char grade[14];
         cout << "Student Number: ";
         cin  >> no;
         cout << "Student Grades: ";
         cin  >> grade;
         cpa[i].set(no, grade);
     }

     for (int i = 0; i < n; i++)
         cpa[i].display();
         cout << endl;

     delete [] cpa;
     cpa = nullptr;
 }
```

Memory Issues

Important issues arise with dynamic memory allocation and deallocation include:

- memory leaks
- insufficient memory

Memory Leak

A memory leak occurs if an application loses the address of dynamically allocated memory that has not deallocated. This may occur if

- a pointer to dynamic memory goes out of scope before the application has deallocated that memory
- a pointer to dynamic memory changes its value before the application has deallocated the memory starting at that value

Memory leaks are difficult to find because they often do not halt execution immediately. We might only become aware of their existence indirectly through progressively slower execution or incorrect results.

Insufficient Memory

Many platforms have sufficient hardware and operating system software to support large allocations of dynamic memory. On those platforms where memory is severly limited, a realistic possibility exists that the operating system might not be able to provide the amount of dynamic memory requested.

If the operating system cannot provide the requested dynamic memory, the application may stop executing. One method of trapping failures to allocate memory is described in the chapter entitled The ISO/IEC Standard.

Single Instances

We can allocate dynamic memory for single instances of any type. The allocation and deallocation syntax is similar to that for arrays.

Allocation

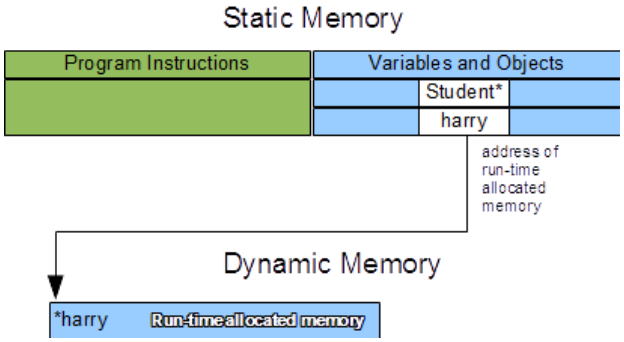
The keyword ***new*** without the brackets allocates dynamic memory for a single variable or object.

A dynamic allocation statement takes the form

```
pointer = new Type;
```

For example, to store one instance of a ***Student*** in dynamic memory, we write

```
Student* harry = nullptr;    // a pointer in static memory
harry = new Student;         // an instance of Student in dynamic memory

// we must deallocate harry later
```



Deallocation

The keyword ***delete*** without the brackets deallocates dynamic memory at the address specified.

A dynamic deallocation statement takes the form

```
delete pointer;
```

delete takes address that was returned by ***new***.

For example, to deallocate the memory for ***harry*** that was allocated dynamically above, we write

```
delete harry;
harry = nullptr;  // good programming style
```

1.6 Check Your Progress

Q.1 What is Dynamic Memory? Explain with the help of an example.
Q.2 Define the following terms.

I. System memory
II. Static Memory

Q.4 What are input and output stream? Explain.
Q.5 What is overloading and Pass By Reference?

Block-3

1.1 Learning Objectives

After going through this unit, the learner will able to:

- Understand the concept of Encapsulation
- Define and learn about Construction and Destruction
- Define the current object
- Learn about classes and resources
- Define member operator
- Learn about helper functions
- Define Custom I/O Operators
- Define Custom File Operators

1.2 Introduction to Encapsulation

In programming languages, **encapsulation** is used to refer to one of two related but distinct notions, and sometimes to the combination thereof:

- A language mechanism for restricting access to some of the object's components.
- A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data.

Some programming language researchers and academics use the first meaning alone or in combination with the second as a distinguishing feature of object-oriented programming, while other programming languages which provide lexical closures view encapsulation as a feature of the language orthogonal to object orientation.

The second definition is motivated by the fact that in many OOP languages hiding of components is not automatic or can be overridden; thus, information hiding is defined as a separate notion by those who prefer the second definition.

The features of encapsulation are supported using classes in most object-oriented programming languages, although other alternatives also exist.

1.3 Construction and Destruction

In object-oriented languages, a *class* is the type that encapsulates state and logic. It describes the structure of the data that objects hold and the rules under which member functions access and change that data. A well-encapsulated class has all implementation details hidden within itself: both its logic and its state structure. Clients communicate with objects of a well-encasulated class only through an interface of public member functions.

This chapter describes some basic class features and the special member functions that initialize and tidy up objects. It covers the order of memory allocation and deallocation during object construction and destruction and overloading of the special function that initializes an object.

Basic Class Features

A class describes how to interpret the data in a region of memory and the rules for operating on that data.

Object or Instance

Each object or instance of a class occupies its own region of memory.

A definition of an object takes the form

```
Type instance;
```

Type denotes the name of the class. ***instance*** denotes the name of the object or instance of the class.

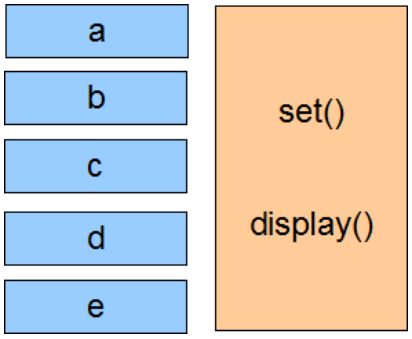
For example, to create an object or instance of our ***Student*** class named ***harry***, we write:

```
Student harry;
```

To create five objects or instances of our Student class, we write:

```
Student a, b, c, d, e;
```

The compiler allocates five regions in static memory, each of which holds the data for one object. Each region contains space for two data members - no and grade. The compiler stores the program instructions contained in the member functions once.



Instance Variables

We call the data members declared in the class definition the object's *instance variables*. Instance variables may (amongst others) be of

- fundamental type (int, double, char, etc.)
- compound type
  - class type (struct, class)
  - pointer type (to instances of data types - fundamental or compound)
  - reference type (to instances of data types - fundamental or compound)

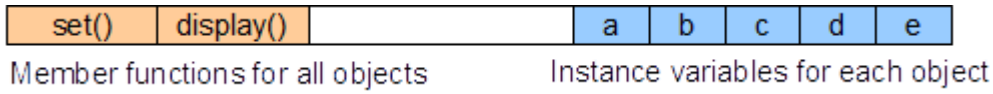
Logic

The logic within the member functions of a class is identical for every instance of the class and there is no need to allocate separate memory for the logic associated with each object. The compiler only stores the instance variables separately. At run-time each call to a member function on an object accesses the same code, while accessing different instance variables - those of the object on which we have called the member function.

For example, calling the same ***display()*** function on five different ***Student*** objects displays five different sets of information in the same way:

```
Student a, b, c, d, e;

// different settings for each object

a.display();  // displays the data stored in a
cout << endl;
b.display();  // displays the data stored in b
cout << endl;
c.display();  // displays the data stored in c
cout << endl;
d.display();  // displays the data stored in d
cout << endl;
e.display();  // displays the data stored in e
cout << endl;
```

The memory allocated by the compiler for member function code and object data is illustrated below:



Class Privacy

C++ implements privacy at the class level. Any member function can access any private member of its class, including any data member of any instance of its class, including any instance other than that on which we have called the member function. In other words, there is no privacy *at the object level*.

For example, we may refer to a private data member of a ***Student*** object within a member function called on another ***Student*** object:

```
class Student {
    int no;
    char grade[14];
public:
    void copyIn(const Student& src);
    void set(int n, const char* g);
    void display() const;
};

// ...

void Student::copyIn(const Student& src) {
    no = src.no; // copy from one object to another
    strcpy(grade, src.grade);
}

// ...

int main() {
    Student harry, backup;
    harry.set(975, "ABA");
    backup.copyIn(harry);
}
```

Here, ***copyIn(const Student& src)*** copies the values from the private data members of ***harry*** into the private data members of ***backup***.

Constructor

Comprehensive encapsulation requires some mechanism for initializing the data members of an object at creation-time. Without initialization, an object's data members contain undefined values until a client calls a modifier on the object and any client can inadvertently 'break' the

object by calling member functions in the wrong order. For instance, a client could call a member function to read a file before having called the member function to open the file.

For example, the following code generates the output on the right

```
// Calling in the Wrong Order
 // Wrong_order.cpp

#include <iostream>
using namespace std;
const int M = 13;

class Student {
   int no;
   char grade[M+1];
public:
   void set(int, const char*);
   void display() const;
};

void Student::set(int n, const char* g) {
   int i;
   bool valid = true; // assume valid input, check invalidity

   if (n < 1)
      valid = false;
   else
      for (i = 0; g[i] != '\0' && valid; i++)
         valid = g[i] >= 'A' && g[i] <= 'F' && g[i] != 'E';

   if (valid) {
      // accept client's data
      no = n;
      for (i = 0; g[i] != '\0' && i < 13; i++)
         grade[i] = g[i];
      grade[i] = '\0';  // set the last byte to the null byte
   }
   else {
      // ignore client's data, set an empty state
      no = 0;
      grade[0] = '\0';
   }
}

void Student::display() const {
   if (no != 0)
      cout << no << ' ' << grade;
   else
      cout << "no data available";
}

int main () {
   Student harry;
```

```
    harry.display();                                    12052848
    cout  << endl;
    harry.set(1234, "ABACA");
    harry.display();
    cout  << endl;                                      1234 ABACA
}
```

Initially the student number of ***harry*** is undefined and the first call to ***display()*** outputs incomprehensible results and may even produce a segmentation fault.

To avoid breaking objects, we initialize their data members to an empty state upon creation and insert dedicated logic for objects in an empty state in each public member function.

Definition

In C++, the special member function that any object calls at creation-time is called a *constructor*. This member function executes preliminary logic and we use it to initialize the object's instance variables.

The constructor takes its name from the class itself. The prototype for a no-argument constructor takes the form

```
Type();
```

Type is the name of the class. A constructor declarations does not include a return data type.

Example

To define a constructor for our Student class, we declare its prototype explicitly in the class definition:

```
class Student {
    int no;
    char grade[M+1];
public:
    Student();
    void set(int, const char*);
    void display() const;
};
```

We define the constructor in the implementation file:

```
Student::Student() {
    no = 0;
    grade[0] = '\0';
}
```

Default Constructor

If we do not declare a constructor in the class definition, the compiler inserts a default no-argument constructor with an empty body:

```
Student::Student() {
}
```

This default constructor leaves the instance variables uninitialized.

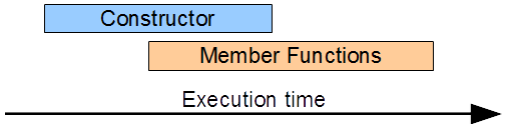
Understanding Order Construction

The compiler constructs an object in the following order

1. allocates memory for each instance variable in the order listed in the class definition
2. executes the logic within the constructor

Member Function Calls

The constructor starts executing before any normal member function is called.



Multiple Objects

If we define multiple objects in a single declaration, the compiler creates them in the order specified by the declaration.

For example, the following code generates the output on the right

```
// Constructors
 // constructors.cpp

 #include <iostream>
 #include <cstring>
 using namespace std;
 const int M = 13;

 class Student {
     int no;
     char grade[M+1];
 public:
     Student();
     void set(int, const char*);
     void display() const;
 };

 // initializes the data members
 //
 Student::Student() {
     cout << "Entering constructor" << endl;
     no = 0;
     grade[0] = '\0';
 }

 void Student::set(int n, const char* g) {
     // see above for complete validation logic
     no = n;
     strncpy(grade, g, M);
 }

 void Student::display() const {
     if (no != 0)
         cout << no << ' ' << grade;
     else
         cout << "no data available";
 }

 int main () {
     Student harry, josee;

     harry.set(1234, "ABACA");
     josee.set(1235, "BBCDC");
     harry.display();
     cout  << endl;
     josee.display();
     cout  << endl;
 }
```

```
Entering constructor
Entering constructor

1234 ABACA

1235 BBCDC
```

The compiler construct ***harry*** first and ***josee*** afterwards.

Safe Empty State

Initializing an object's instance variables in a constructor ensures that the object has a well-defined state from the *instant of its creation*. In the above example, we say that ***harry*** and ***josee*** are in *safe empty state*s until the set() member function changes their states. If our code calls member functions on these objects in any 'unusual' order, the objects do not break and the results are still as expected.

For example,

```
// Safe Empty State
 // safeEmpty.cpp

 #include <iostream>
 using namespace std;

 int main ( ) {
     Student harry, josee;

     harry.display();
     cout  << endl;
     josee.display();
     cout  << endl;
     harry.set(1234,"ABACA");
     josee.set(1235,"BBCDA");
     harry.display();
     cout  << endl;
     josee.display();
     cout  << endl;
 }
```

```
 Entering constructor
 Entering constructor
 0

 0


 1234 ABACA

 1235 BBCDC
```

The initial values displayed for each object are their safe empty state values.
The safe empty state value is identical for all objects of the same class.

Destructor

Comprehensive encapsulation also requires some mechanism for tidying up just before the end of an object's lifetime. An object that has written data to a file may need to flush the file's buffer before the object going out of scope. An object that has allocated memory dynamically may need to deallocate that memory before going out of scope. C++ lets us define a special member function called the *destructor* that executes automatically at the point of an object's destruction.

Definition

In C++, the special member function that every object calls just before the end of its lifetime is called the *destructor*. We code this member function with the terminal logic.

The destructor takes its name from the class itself and prefixes that name with the tilde symbol (~). The prototype for a destructor takes the form

```
~Type( );
```

Type is the name of the class. A destructor does not have any parameters, does not return a value and does not have a return data type.

An object's destructor

- is called automatically
- cannot be overloaded
- cannot be called explicitly

Example

To define the destructor for our ***Student*** class, we declare its prototype in the class definition:

```
class Student {
     int no;
     char grade[M+1];
 public:
     Student();
     ~Student();
     void set(int, const char*);
     void display() const;
 };
```

and define the member function in the implementation file:

```
Student::~Student() {
    // insert our terminal code here
 }
```

Default Destructor

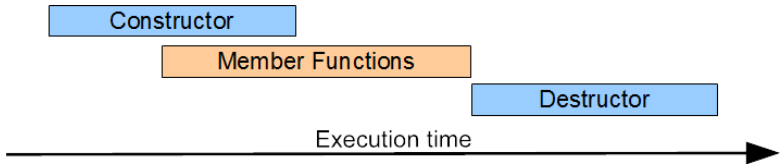
If we don't declare a destructor in the class definition, the compiler defines the destructor with an empty body:

```
Student::~Student() {
 }
```

Understanding Order

Member Function Calls

The object's destructor starts executing only after every normal member function has completed its execution.



The object cannot call any member function after having called its destructor.

Destruction

The compiler destroys an object in the following order

1. executes the logic of its destructor
2. deallocates memory for each instance variable in opposite order to that listed in the class definition

Multiple Objects

The compiler destroys objects in opposite order to the order of their creation.

For example, the following code generates the output on the right:

```
// Constructors and Destructors
 // destructors.cpp

 #include <iostream>
 #include <cstring>
 using namespace std;
 const int M = 13;

 class Student {
     int no;
     char grade[M+1];
 public:
     Student();
     ~Student();
     void set(int n, const char* g);
     void display() const;
 };

 Student::Student() {
     cout << "Entering constructor" <<
```

```
endl;
     no = 0;
     grade[0] = '\0';
 }

 // executed before object goes out of
scope
 //
 Student::~Student() {
    cout << "Entering destructor for " << no
         << endl;
 }

 void Student::set(int n, const char* g){
     // see above for validation logic
     no = n;
     strcpy(grade, g);
 }

 void Student::display() const {
     cout << no << ' ' << grade;
 }

 int main () {
     Student harry, josee;

     harry.set(1234, "ABACA");
     josee.set(1235, "BBCDC");
     harry.display();
     cout  << endl;
     josee.display();
     cout  << endl;
 }
```

```
 Entering constructor
 Entering constructor

 1234 ABACA

 1235 BBCDC


 Entering destructor for
1235
 Entering destructor for
1234
```

The compiler destroys ***josee*** first and ***harry*** last.

Construction and Destruction of Arrays

The order of constructing and destroying the elements of an array of objects follows directly from the order described above.

The compiler creates the elements of an array one at a time from its first element to its last. Each object calls the no-argument constructor at creation-time. At deallocation, the compiler destroys the last element first and proceeds sequentially through the array until it destroys the first element last.

For example, the following code generates the output on the right:

```
// Constructors, Destructors and Arrays
 // ctorsDtorsArrays.cpp

 #include <iostream>
 #include <cstring>
 using namespace std;
 const int M = 13;

 class Student {
     int no;
     char grade[M+1];
 public:
     Student();
     ~Student();
     void set(int, const char*);
     void display() const;
 };

 Student::Student() {
     cout << "Entering constructor" <<
endl;
     no = 0;
     grade[0] = '\0';
 }

 Student::~Student() {
     cout << "Entering destructor for " <<
          no << endl;
 }

 void Student::set(int n, const char* str){
     // see above for validation logic
     // code
     no = n;
     strcpy(grade, g);
 }

 void Student::display() const {
     cout << no << ' ' << grade;
 }

 int main () {
     Student a[3];

     a[0].set(1000, "AAAAA");
     a[2].set(1002, "CCCCC");
     a[1].set(1001, "BBBBB");
     for (int i = 0; i < 3; i++) {
         a[i].display();
         cout  << endl;
     }
 }
```

```
 Entering constructor
 Entering constructor
 Entering constructor
 1000 AAAAA

 1002 CCCCC

 1001 BBBBB


 Entering destructor for
1002
 Entering destructor for 1001
 Entering destructor for 1000
```

The destructor for element ***a[2]*** executes before the destructor for ***a[1],*** which executes before the destructor for ***a[0].*** Note that the order of destruction is based on order of construction and not on order of usage

Overloading Constructors

Overloading a class' constructor adds communication options for clients. A client can select the most appropriate set of arguments to specify at creation time.

For example, to let a client initialize a Student object with a student number and a set of grades, we define a two-argument constructor with one int parameter and one const char* parameter:

```
// Two-Argument Constructor
 // overload.cpp

 #include <iostream>
 using namespace std;
 const int M = 13;

 class Student {
     int no;
     char grade[M+1];
 public:
     Student();
     Student(int, const char*);
     ~Student();
     void display() const;
 };

 Student::Student() {
     cout << "Entering 0-arg constructor" <<
endl;
     no = 0;
     grade[0] = '\0';
 }

 Student::Student(int n, const char* g) {
     cout << "Entering 2-arg constructor" <<
endl;
     int i;
     bool valid = true; // assume valid input

     if (n < 1)
         valid = false;
     else
         for (i = 0; g[i] != '\0' && valid;
i++)
             valid = g[i] >= 'A' && g[i] <=
'F'
              && g[i] != 'E';
```

```
    if (valid) {
        // accept client's data
        no = n;
        for (i = 0; g[i] != '\0' && i < 13;
i++)
            grade[i] = g[i];
        grade[i] = '\0';  // set last byte to
null
    }
    else {
        // ignore client's data, set empty
state
        no = 0;
        grade[0] = '\0';
    }
 }

 Student::~Student() {
    cout << "Entering destructor for " <<
         no << endl;
 }

 void Student::display() const {
    cout << no << ' ' << grade;
 }

 int main () {
    Student harry(1234,"ABACA"),
            josee(1235,"BBCDA");

    harry.display();
    cout  << endl;
    josee.display();
    cout  << endl;
 }
```

```
 Entering 2-arg
constructor
 Entering 2-arg
constructor

 1234 ABACA

 1235 BBCDA
 Entering destructor for
1235
 Entering destructor for
1234
```

This new constructor includes all of our validation logic. The compiler calls one and only one constructor at creation. In this example, the compiler does not call the no-argument constructor.

Note that we have replaced the set() member function with the two-argument constructor.

No-argument constructor is not always implemented

If the class definition includes the prototype for a constructor with some parameters but does not include the prototype for a no-argument constructor, the compiler DOES NOT insert an empty-body, no-argument constructor. The compiler only inserts an empty-body, no-argument constructor if the class definition does not include a prototype for ANY constructor.

If we define a constructor with some parameters, we typically also define a no-argument constructor. The creation of arrays of objects requires a no-argument constructor (each element of the array calls the no-argument constructor at creation time).

1.4 The Current Object

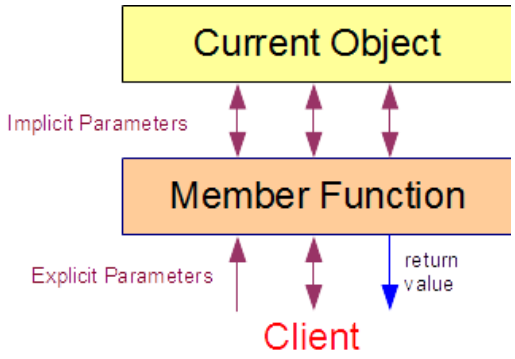
Every member function executes on a specific object; that is, on one particular set of instance variables. That object is the object on which the client has called the member function. We refer to the object as the *current object* for that member function. In other words, the current object is the region of memory containing the data on which a member function is currently operating.

This chapter describes the mechanism by which a member function accesses the current object and shows how to refer to the current object from within the member function.

Member Function Access

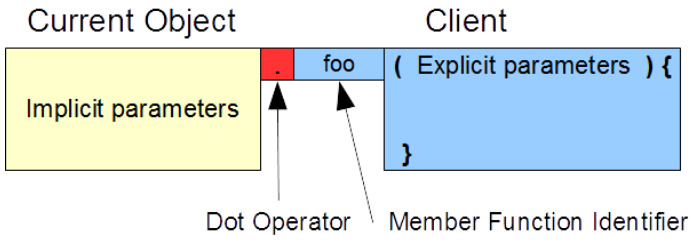
A member function accesses non-local information through parameters and returns information to its caller through parameters and possibly a return value. A member function's parameters are of two distinct kinds:

- explicit - communicate with the client code
- implicit - communicate with the instance variables



Explicit parameters receive information from the client and return information to the client. We define them explicitly in the function header. Their lifetime is the period during which the member function is in control of execution.

Implicit parameters the member function to the current object.



The syntax of a call to a normal member function reflects this mechanism. The name of the object on which the client calls this function represents the implicit parameters, while the arguments that the client passes to the function initialize the explicit parameters.

Consider the constructors and the calls to the ***display()*** member function in the following code snippet:

```
// ...

Student::Student(int n, const char* g) {
    no = n;
    strcpy(grade, g);
}

void Student::display() const {
    cout << no << ' ' << grade;
}

// ...

int main ( ) {
    Student harry(1234, "ABACA"), josee(1235, "BBCDA");

    harry.display();
    cout << endl;
    josee.display();
    cout << endl;
}
```

```
1234 ABACA

1235 BBCDA
```

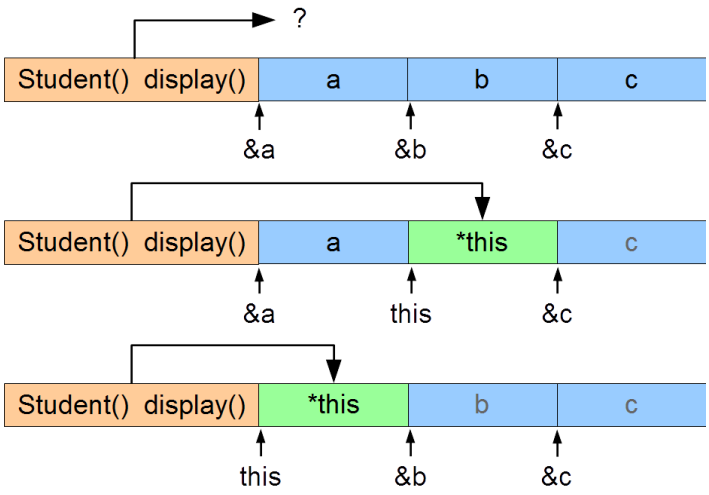
The constructor for harry receives data in its explicit parameters and copies that data through its implicit parameters into the instance variables of its current object. The constructor for josee receives data in its explicit parameters and copies that data through its implicit parameters into the instance variables of its current object.

The first call to the ***display()*** member function accesses ***harry*** through its implicit parameters. The second callaccesses ***josee*** through its implicit parameters.

This

The complete set of instance variables associated with the current object has its own address. The keyword this returns ***this*** address. That is, ***this*** holds the address of the region of memory that contains all of the data stored in the current object. ****this*** refers to the current object itself; that is, to the entire set of its instance variables.

We use this keyword within a member function to refer to the set of instance variables that the member function is currently accessing through its implicit parameters.



For example, to upgrade the ***display()*** member function to return a copy of the object upon which it has been called, we write:

```
Student Student::display() const {
     cout << no << ' ' << grade;
     return *this;
 }
 int main() {
     Student harry(1234,"ABACA"), backup;
     backup = harry.display();
     cout << endl;
     backup.display();
     cout << endl;
 }
```

```
1234 ABACA
1234 ABACA
```

The keyword ***this*** is only accessible from within a member function and has no meaning outside member functions.

Reference to the Current Object

We can upgrade our definition of ***display()*** by returning an unmodifiable reference to the current object rather than a copy of the object. This would makes a difference if the object was large, since copying all of its instance variables would be compute intensive. Returning a reference only involves copying the object's address, which is typically a 4-byte operation:

```
const Student& Student::display() const {

     cout << no << ' ' << grade;

     return *this;
 }
```

The ***const*** qualifier on the return type prohibits client code from placing the call to the member function on the left side of an assignment operator and thereby allowing a change to the instance variables themselves.

Assigning to the Current Object

To copy an object's instance variables into the current object, we dereference the keyword and use *this as the left operand in an assignment expression:

```
*this =             ;
```

Example - Extracting Input Data

Let us introduce a member function to our Student class called read() that

- extracts data from standard input
- stores that data in the current object only if the data is valid
- leaves the current object unchanged if the data is invalid

To avoid duplicating validation logic, we

- construct a local Student object passing the input data to the two-argument constructor
- check the student number to determine if the local object accepted the data
- assign the local object to the current object if the data is valid

```
void Student::read() {

     int no;          // will hold the student number
     char grade[14]; // will hold the grades

     cout << "Enter student number : ";
     cin >> no;
     cin.ignore(); // remove newline character
     cout << "Enter student grades : ";
     cin.getline(grade, 14);

     // construct the local object
     Student temp(no, grade);
     // if data is valid student number is non-zero
     if (temp.no != 0)
         // copy the local object into the current object
```

```
        *this = temp;
}
```

Since the local object ***(temp)*** and the current object are instances of the same class, this member function can access each of the local object's instance variables directly.

1.5 Classes and Resources

We design and code classes independently of their client applications. In cases where a client determines the amount of memory that an object requires, we cannot specify the memory requirements at compile-time and must postpone allocation of that memory until run-time. Only once the client starts instantiating the object does that object know how much memory the client requires. To review run-time memory allocation and deallocation see the chapter entitled Dynamic Memory.

Memory that an object allocates at run-time represents a resource of that object's class. The management of this resource requires additional logic that was unnecessary for simpler classes that do not access resources. This additional logic ensures proper handling of the resource and is often called deep copying and assignment.

This chapter describes how to implement deep copying and deep assignment logic. The member functions that manage resources are the constructors, the assignment operator and the destructor.

Resource Instance Pointers

A C++ object refers to a resource through a *resource instance pointer*. This pointer holds the address of the resource. The address (that is, the resource) lies outside the object's static memory.

Case Study

Let us upgrade our Student class to accomodate a variable number of grades. The client specifies the number at run-time. The grades are now a resource. We allocate

- static memory for the resource instance variable (grade)
- dynamic memory for the grade string itself

In this section, we focus on the constructors and the destructor for our Student class. The client does not copy or assign and we postpone the copying and assignment logic to later sections:

```
// Resources - Constructor and Destructor
 // resources.cpp

 #include <iostream>
 #include <cstring>
 using namespace std;

 class Student {
     int no;
     char* grade;
 public:
     Student();
     Student(int, const char*);
     ~Student();
     void display() const;
 };

 Student::Student() {
     no = 0;
     grade = nullptr;
 }

 Student::Student(int n, const char* g) {
     int i;
     bool valid = true; // assume valid input, check
invalidity

     // validate client data
     if (n < 1)
         valid = false;
     else {
         for (i = 0; g[i] != '\0' && valid; i++)
             valid = g[i] >= 'A' && g[i] <= 'F' && g[i] !=
'E';

     if (valid) {
         // accept client data
         // allocate dynamic memory
         no = n;
         grade = new char[strlen(g) + 1];
         strcpy(grade, g);
     }
     else {
         // set to a safe empty state
         no = 0;
         grade = nullptr;
     }
 }

 Student::~Student() {
     // deallocate previously allocated memory
     delete [] grade;
 }
```

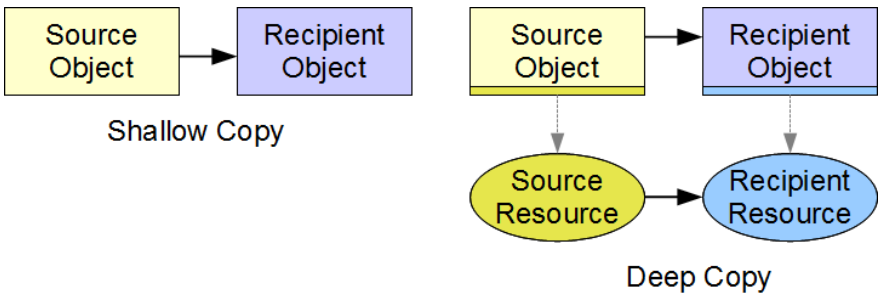
```
void Student::display() const {
    cout << no << ' ' <<
         ((grade != nullptr) ? grade : "");
}

int main ( ) {
    Student harry(1234, "ABACA");                                    1234
                                                                     ABACA
    harry.display();
    cout << endl;
}
```

The no-argument constructor places the object in a safe empty state. The two-argument constructor allocates dynamic memory for the resource only if the data received is valid. The conditional expression in the display() query distinguishes the safe empty state. The destructor deallocates any memory that the constructor allocated. Deallocating memory at the nullptr address has no effect.

Deep Copies and Assignments

In a typical class design involving resources, we expect each resource associated with one object to be independent of the resource associated with another. That is, if we change the resource data in one object, we expect the resource data in the other object to remain unchanged. In copying and assigning objects we ensure this resource independence through deep copies and deep assignments. Deep copies and assignments involve copying the resource data. Shallow copies and assignments, which involve copying the instance variables only, are only approriate for class without resources.

Implementing deep copying and assignment logic requires separate allocation of memory. The resource instance pointer in each object points to a different location in dynamic memory.



For each deep copy, we allocate memory for a new resource and copy the contents of the original resource into that new memory. We shallow copy only those instance variables that are NOT

resource instance variables. For example, in our ***Student*** class, we shallow copy the student number, but not the address stored in the ***grade*** pointer.

The two special member functions that manage allocations and deallocations associated with deep copies and assignments are:

- the copy constructor
- the copy assignment operator

If we do not declare a copy constructor, the compiler inserts code that implements a shallow copy. If we do not declare an assignment operator, the compiler inserts code that implements a shallow assignment.

Copy Constructor

The copy constructor defines the logic for copying from a source object to a *newly created* object of the same type. The compiler calls this constructor whenever we

1. initialize an object at creation
2. copy an object by value in a function call
3. return an object by value from a function

Declaration

The declaration of a copy constructor takes the form

```
Type(const Type&);
```

where ***Type*** is the name of the class

For example, we insert the declaration into the definition of our Student class:

```
// Student.h

class Student {
    int no;
    char* grade;
public:
    Student();
    Student(int, const char*);
    Student(const Student&);
    ~Student();
    void display() const;
};
```

Definition

The definition of a copy constructor contains logic that

1. performs a shallow copy on all of the non-resource instance variables
2. allocates memory for each new resource
3. copies data from the source resource(s) to the newly created resource(s)

For example, the following code performs a deep copy on objects of our Student class:

```
// Student.cpp

 #include <iostream>
 #include <cstring>
 using namespace std;
 #include "Student.h"

 // ...

 Student::Student(const Student& src) {

     // shallow copy
     no = src.no;

     // allocate dynamic memory for grade string
     if (src.grade != nullptr) {
         grade = new char[strlen(src.grade) + 1];
         // copy data from the original resource
         // to newly allocated resource
         strcpy(grade, src.grade);
     }
     else
         grade = nullptr;
 }
```

Since the source data has been validated on original receipt from the client and privacy constraints ensure that this data has not been corrupted in the interim, we do not need to validate the data in the copy constructor.

Assignment Operator

The assignment operator defines the logic for copying data from an existing object to another *existing* object. The compiler calls this member operator whenever we write expressions of the form

```
identifier = identifier
```

where ***identifier*** refers to the name of an object.

Declaration

The declaration of an assignment operator takes the form

```
Type& operator=(const Type&);
```

where the left ***Type*** is the return type and the right ***Type*** is the type of the source operand.

For example, we insert the declaration into the definition of our ***Student*** class:

```
// Student.h

class Student {
    int no;
    char* grade;
public:
    Student();
    Student(int, const char*);
    Student(const Student&);
    Student& operator=(const Student&);
    ~Student();
    void display() const;
};
```

Definition

The definition of the assignment operator contains logic that:

- checks for self-assignment
- shallow copies the non-resource instance variables to destination variables
- deallocates any previously allocated resource for the current object
- allocate a new resource for the current object
- copies resource data associated with the source object to the newly allocated resource memory of the current object

For example, the following code performs a deep assignment on objects of our ***Student*** class:

```
// Student.cpp

// ...

Student& Student::operator=(const Student& source) {

    // check for self-assignment
    if (this != &source) {
        // shallow copy non-resource variable
```

```
        no = source.no;
        // deallocate previously allocated dynamic memory
        delete [] grade;
        // allocate new dynamic memory, if needed
        if (source.grade != nullptr) {
            grade = new char[strlen(source.grade) + 1];
            // copy the resource data
            strcpy(grade, source.grade);
        }
        else
            grade = nullptr;
    }
    return *this;
}
```

To trap a self-assignment ***(a = a),*** we compare the address of the current object to the address of the source object. If the addresses match, we skip the assignment logic altogether. If we neglected to check for self-assignment, the deallocation statement would deallocate the memory holding the resource data and we would lose access to the resource resulting in our logic failing at the call to ***std::strlen()***.

Localization

The code in our definition of the copy constructor is identical to most of the code in our definition of the assignment operator. To avoid duplication we can:

- localize the common code in a private member function and call that member function from the copy constructor and the assignment operator
- call the assignment operator directly from the copy constructor

Private Member Function

In the following example, we localize the common code is in a private member function named ***init()*** and call this function from our copy constructor and assignment operator call:

```
void Student::init(const Student& source) {
    no = source.no;
    if (source.grade != nullptr) {
        grade = new char[strlen(source.grade) + 1];
        strcpy(grade, source.grade);
    }
    else
        grade = nullptr;
}

Student::Student(const Student& source) {
    init(source);
}
```

```
 Student& Student::operator=(const Student& source) {
     if (this != &source) {  // check for self-assignment
         // deallocate previously allocated dynamic memory
         delete [] grade;
         init(source);
     }
     return *this;
 }
```

Direct Call

In the following example, we initialize the resource instance variable in the copy constructor to ***nullptr*** and call the assignment operator directly:

```
Student::Student(const Student& source) {
     grade = nullptr;
     *this = source; // calls assignment operator
 }
```

```
 Student& Student::operator=(const Student& source) {
     if (this != &source) {  // check for self-assignment
         no = source.no;
         // deallocate previously allocated dynamic memory
         delete [] grade;
         // allocate new dynamic memory
         if (source.grade != nullptr) {
             grade = new char[strlen(source.grade) + 1];
             // copy resource data
             strcpy(grade, source.grade);
         }
         else
             grade = nullptr;
     }
     return *this;
 }
```

Assigning ***grade*** to ***nullptr*** in the copy constructor ensures that the assignment operator does not deallocate any memory if called by the copy constructor.

Copies Prohibited

Certain class designs may require that we prohibit any client from copying any instance of a class. To prohibit copying and/or assigning, we declare the copy constructor and/or the assignment operator as deleted members of our class:

```
class Student {
     int no;
     char* grade;
 public:
     Student();
```

```
    Student(int, const char*);
    Student(const Student& source) = delete;
    Student& operator=(const Student& source) = delete;
    ~Student();
    void display() const;
};
```

Use of the keyword ***delete*** in this context has no relation to its use in deallocating dynamic memory.

1.6 Member Operators

In programming languages, an expression consists of an operator and a set of operands. The expression evaluates to a value of its own type. In C++, all operators are built into the core language. The core language defines the logic for operands of fundamental type and the logic for the assigning one object to another of the same type. To define expressions for operand of class type overloading of the operator for the class type. That is, in order for a client to evaluate an expression involving an object of class type, we need to overload the operator for an operand of that type. Overloading an operator simply entails adding a function to the class definition to specify the logic of the operation.

This chapter lists the operators that C++ lets us overload and describes how to define member functions that overload operators. The chapter covers unary and binary operations on the current object. Unary operations involve an operator and one operand, while binary operations involve an operator and two operands. The description includes how to define casting operators using single-argument constructors.

Operations

C++ identifies an overloaded operation by the keyword operator the operator's symbol and the type(s) of the operand(s) in the expression. The signature of a member function that overloads the operator for an operand of class type includes the symbol and the type of its right operand, if any. The left operand is the current object.

Candidates for Overloading

C++ lets us overload the following operators (amongst others):

- binary arithmetic (+ - * / %)
- assignment (= += -= *= /= %=)
- unary pre-fix post-fix plus minus (++ -- + -)

- relational (== < > <= >= !=)
- logical (&& || !)
- insertion, extraction (<< >>)

C++ DOES NOT ALLOW overloading of the following operators (amongst others):

- the scope resolution operator (::)
- the member selection operator (.)
- the member selection through pointer to member operator (.*)
- the conditional operator (?:)

C++ DOES NOT let us introduce or define new operators.

Classifying Operators

We classify operators group by the number of operands that they take:

- unary - post-fix increment/decrement, pre-fix increment/decrement, pre-fix plus, pre-fix minus
- binary - assignment, compound assignment, arithmetic, relational, logical
- ternary - conditional operator

Members and Helpers

We overload operators in either of two ways, as:

- member operators - part of the definition of the class
- helper operators - outside the definition of the class

We define operators that change the state of their left operand as member operators and operators that do not change the state of their operands as helper operators. The next chapter covers helper operators.

Overloading a Member Operator

The header of a function that overloads a member operator consists of:

- the return data type
- the keyword ***operator***
- the operator symbol
- function parantheses

- the operand type, if binary

The return type identifies the type of the evaluated expression.

For example, to overload the assignment operator for a ***Student*** to receive a reference to an unmodifiable ***Student*** object as the right operand, we insert the following prototype into the class definition:

```
class Student {
     // ...
     Student& operator=(const Student&);
     // ...
 };

 // ...

 Student harry(975, "BCADB");
 Student backup;
 backup = harry;  // calls the overloaded operator
```

The keyword-symbol combination (operator =) identifies the member function uniquely.

Signature

Every overloaded member operator has its own signature consisting of:

- the operator keyword
- the operation symbol
- the type of its right operand, if any
- the const status of the operation

The compiler evaluates expressions consisting of the operator and the operand type by calling the member function with the signature that matches the operator symbol, the operand type and the ***const*** status.

Promotion or Narrowing of Arguments

If the compiler cannot find an exact match for an operation's signature, the compiler will attempt a rather complicated selection process to find an optimal fit, promoting or narrowing the operand value into a related type if necessary.

Good Design Practice

Programmers expect an operator to perform its operation in a way similar if not identical to the way that the operator performs on any fundamental type as defined by the core language. +

implies addition of two values in a binary operation (not subtraction). In defining an member operator we code its logic to be consistent with that of other types.

Binary Operations

A binary operation consists of one operator and two operands. The left operand is the current object. The operator takes one explicit parameter: the right operand.

The header for a binary member operator takes the form

```
Type operator symbol (type identifier)
```

where ***Type*** is the type of the evaluated expression. ***operator*** identifies some operation. ***symbol*** specifies the operation. ***type*** is the type of the right operand. ***identifier*** is the right operand's name.

For example, let us overload the += operator for a ***char*** as the right operand, in order to add a single grade to a ***Student*** object:

```
// Overloading Operators
 // operators.cpp

 #include <iostream>
 #include <cstring>
 using namespace std;
 const int M = 13;

 class Student {
     int no;
     char grade[M+1];
 public:
     Student();
     Student(int, const char*);
     void display() const;
     Student& operator+=(char g);
 };

 Student::Student() {
     no = 0;
     grade[0] = '\0';
 }

 Student::Student(int n, const char* g) {
     // see Current Object chapter for validation logic
     no = n;
     strcpy(grade, g);
 }

 void Student::display() const {
     cout << no << ' ' << grade << endl;
 }
```

```
Student& Student::operator+=(char g) {
    int i = strlen(grade);
    if (i < M) {
        // add validation logic here
        grade[i++] = g;
        grade[i] = '\0';
    }
    return *this;
}

int main () {
    Student harry(975,"BCADB");

    harry.display();                                    975 BCADB
    harry += 'B';
    harry.display();                                    975 BCADBB
}
```

Unary Operations

A unary operation consists of one operator and one operand. The left operand is the current object. The operator does not take any explicit parameters (with one exception - see post-fix operators below).

The header for a unary member operator takes the form

```
Type operator symbol()
```

where ***Type*** is the type of the evaluated expression. ***operator*** identifies an operation. ***symbol*** identifies the kind of operation.

Pre-Fix Operators

We overload the pre-fix increment/decrement operators to increment/decrement the current object and return the updated value. The header for a pre-fix operator takes the form

```
Type& operator++()  or  Type& operator--()
```

For example, let us overload the pre-fix increment operator for our ***Student*** class so that a pre-fix expression increases all of the ***Student's*** grades by one grade letter, if possible:

```
// Pre-Fix Operators
 // preFixOps.cpp

#include <iostream>
#include <cstring>
using namespace std;
const int M = 13;

class Student {
    int no;
    char grade[M+1];
```

```
public:
    Student();
    Student(int, const char*);
    void display() const;
    Student& operator++();
};

Student::Student() {
    no = 0;
    grade[0] = '\0';
}

Student::Student(int n, const char* g) {
    // see Current Object chapter for validation logic
    no = n;
    strcpy(grade, g);
}

void Student::display() const {
    cout << no << ' ' << grade << endl;
}

Student& Student::operator++() {
    for (int i = 0; grade[i] != '\0'; i++)
        if (grade[i] == 'F') grade[i] = 'D';
        else if (grade[i] != 'A') grade[i]--;
    return *this;
}

int main () {
    Student harry(975,"BCADB");
    harry.display();                                    975 BCADB
    backup = ++harry;
    harry.display();                                    975 ABACA
}
```

Post-Fix Operators

We overload the post-fix operators to increment/decrement the current object *after* returning its value. The header for a post-fix operator takes the form

***Type* operator++(int) or *Type* operator--(int)**

The ***int*** type in the header distinguishes the post-fix operators from their pre-fix counterparts.

For example, let us overload the incrementing post-fix operator for our Student class so that a post-fix expression increases all of the Student's grades by one grade letter, if possible:

```
// Post-Fix Operators
 // postFixOps.cpp

 #include <iostream>
 #include <cstring>
 using namespace std;
```

```
const int M = 13;

class Student {
    int no;
    char grade[M+1];
public:
    Student();
    Student(int, const char*);
    void display() const;
    Student& operator++();
    Student operator++(int);
};

Student::Student() {
    no = 0;
    grade[0] = '\0';
}

Student::Student(int n, const char* g) {
    // see Current Object chapter for validation logic
    no = n;
    strcpy(grade, g);
}

void Student::display() const {
    cout << no << ' ' << grade << endl;
}

Student& Student::operator++() {
    for (int i = 0; grade[i] != '\0'; i++)
        if (grade[i] == 'F') grade[i] = 'D';
        else if (grade[i] != 'A') grade[i]--;
    return *this;
}

Student Student::operator++(int) {
    Student s = *this;  // save the original
    ++(*this);          // call the pre-fix operator
    return s;           // return the original
}

int main () {
    Student harry(975,"BCADB");
    Student backup;
    harry.display();                                    975 BCADB
    backup = harry++;
    backup.display();                                   975 BCADB
    harry.display();                                    975 ABACA
}
```

We avoid duplicating logic by calling the pre-fix operator from the post-fix operator.

The return values of the pre-fix and post-fix operators differ. The post-fix operator returns a copy of the current object as it was *before* any changes took effect. The pre-fix operator returns a reference to the current object *after* the changes have taken effect.

Type Conversions

Member operators include type conversion operators, which define implicit conversions to different types, including fundamental types.

For the following code to compile, the compiler must know how to convert a ***Student*** object into a ***bool*** value:

```
Student harry;

 if (harry)
     harry.display();
```

To this effect, we define a conversion operator that returns ***true*** if the ***Student*** object has valid data and ***false*** if the object is in a safe empty state.

We add its declaration to the class definition

```
const int M = 13;

 class Student {
     int no;
     char grade[M+1];
 public:
     Student();
     Student(int, const char*);
     void display() const;
     operator bool() const;
 };
```

and define the conversion operator in the implementation file

```
#include "Student.h"

 // ...

 Student::operator bool() const { return no != 0; }
```

Design Consideration

Conversion operators easily lead to ambiguities. Good design uses them quite sparingly and keeps their implementations trivial.

Single-Argument Constructors

A single-argument constructor defines the rule for promoting the value of its parameter to its class type. This type of constructor defines not only how to construct an object using only a

single argument but also how to convert an argument of that type into an object of the class' type.

.

The following program demonstrates both uses of a single-argument constructor that has been overloaded to receive an ***int*** argument:

```
// One-Argument Constructor
 // oneArgCtor.cpp

 #include <iostream>
 using namespace std;
 const int M = 13;

 class Student {
     int no;
     char grade[M+1];
 public:
     Student();
     Student(int);
     Student(int, const char*);
     void display() const;
 };

 Student::Student() {
     no = 0;
     grade[0] = '\0';
 }

 Student::Student(int n) {
     no = n;
     grade[0] = '\0';
 }

 Student::Student(int n, const char* g) {
     // see Current Object chapter for validation logic
     no = n;
     strcpy(grade, g);
 }

 void Student::display() const {
     cout << no << ' ' << grade;
 }

 int main () {
     Student harry(975), nancy;

     harry.display();                                                    975
     cout  << endl;
     nancy = (Student)428;
     nancy.display();                                                    428
     cout  << endl;
 }
```

The first use converts ***975*** to the ***Student*** object ***harry***. The second use casts ***428*** to a ***Student*** object containing the number ***428***. Each resultant object holds an empty grade list.

Promotion

For the same result we could omit the cast and let the compiler promote the ***428*** to a ***Student*** object in the assignment itself:

```
int main () {
     Student harry(975), nancy;

     harry.display();
     cout  << endl;
     nancy = 428; // promotes an int to a Student
     nancy.display();
     cout  << endl;
 }
```

```
975
428
```

The compiler inserts code that creates a temporary ***Student*** object using the single-argument constructor. The constructor receives the value ***428*** and initializes ***no*** to ***428*** and ***grade*** to an empty string. Then, the assignment operator performs a shallow copy from the temporary object to ***nancy***. Finally, the compiler inserts code that destroys the temporary object and removes it from memory.

Explicit

Limiting the number of single-argument constructors in a class definition avoids potential ambiguities in automatic conversions of one data type to another.

To prohibit the compiler from using a single-argument constructor for any implicit conversion, we declare the constructor explicit:

```
class Student {
     int no;
     char grade[M+1];
 public:
     Student();
     explicit Student(int);
     Student(int, const char*);
     void display() const;
 };
```

The second invocation in the example above (**nancy = *428***) would generate a compiler error under this class definition.

Temporary Objects

A temporary object has no name and goes out of scope at the end of same statement as the one within which it is created. For example,

```
int main () {
     Student harry(975), nancy;
```

```
    harry.display();                                                 975
    cout  << endl;
    nancy = Student(428); // temporary Student object
    nancy.display();                                                 428
    cout  << endl;
}
```

A temporary object provides a compact way of calling the constructor on the object's type.

Localizing Constructor Logic

We use temporary objects to localize the validation for a class within one constructor. We use the keyword for the current object ***(*this)*** in assigning the temporary object to the current object:

```
// Localized Validation
 // localize.cpp

 #include <iostream>
 using namespace std;
 const int M = 13;

 class Student {
     int no;
     char grade[M+1];
 public:
     Student();
     Student(int);
     Student(int, const char*);
     void display() const;
 };

 Student::Student() {
    // safe empty state
    no = 0;
    grade[0] = '\0';
 }

 Student::Student(int n) {
    *this = Student(n, ""); // assign temporary to current
 }

 Student::Student(int n, const char* g) {
     int i;
     bool valid = true; // assume valid input, check
invalidity

     // validate client data
     if (n < 1)
         valid = false;
     else
         for (i = 0; g[i] != '\0' && valid; i++)
             valid = g[i] >= 'A' && g[i] <= 'F' && g[i] !=
'E';

     if (valid) {
```

```
        // accept client data
        no = n;
        for (i = 0; g[i] != '\0' && i < M; i++)
            grade[i] = g[i];
        grade[i] = '\0';  // set the last byte to the null
byte
    }
    else {
        // set to a safe empty state
        *this = Student(); // assign temporary to current
    }
 }

 void Student::display() const {
    cout << no << ' ' << grade;
 }

 int main () {                                                    1234
    Student harry(1234,"ABACA"), josee(1235), empty;             ABACA

    harry.display();                                             1235
    cout << endl;
    josee.display();                                             0
    cout << endl;
    empty.display();
    cout << endl;
 }
```

The two-argument constructor creates a temporary object in a safe empty state if the validation fails and assigns that temporary object to the current object.

The single-argument constructor creates a temporary object using the two-argument constructor, which validates the data, and then assigns the temporary object to the current object.

Classes with Resources

Assigning a temporary object to the current object requires additional code if the object manages resources. To prevent the assignment operator from releasing not-as-yet-acquired resources we initialize all resource instance variables to empty values (***nullptr***).

For example, if our Student object stores its grades in dynamic memory, then we need to add the highlighted code:

```
class Student {
    int no;
    char* grade;
 public:
    // ...
 };

 Student::Student() {
   // safe empty state
   no = 0;
   grade = nullptr;
```

```
}

Student::Student(int n) {
   grade = nullptr;
   *this = Student(n, "");
}

Student::Student(int n, const char* g) {
    int i;
    bool valid = true; // assume valid input, check invalidity

    // validate client data
    if (n < 1)
        valid = false;
    else {
        for (i = 0; g[i] != '\0' && valid; i++)
            valid = g[i] >= 'A' && g[i] <= 'F' && g[i] != 'E';

    if (valid) {
        // accept client data
        no = n;
        grade = new char[std::strlen(g) + 1];
        std::strcpy(grade, g);
    }
    else {
        // set to a safe empty state
        grade = nullptr;
        *this = Student();
    }
}
```

Good Design Tip

Using temporary objects to avoid repeating logic is good programming practice. If we update the logic later, there is no chance that we will update the logic in one part of the source code and neglect to update the same logic in another part of the code.

1.7 Helper Functions

A well-encapsulated class can accept external support in the form of global functions that contain additional logic. We call these supporting functions *helper functions*. They access objects solely through their parameters, all of which are explicit. Since helpers are not members of any class, they have no implicit parameters. In a typical helper function at least one parameter receives an object of the class that the function is supporting.

This chapter describes how to define helper functions, including operators, and how to grant select helpers privileged access to the private members of a class.

Free Helpers

A free helper function is a function that does not need access to the private members of the class that it supports. Public member functions on the object provide whatever information the helper function requires. Coupling between a free helper function and the supported class is minimal.

Comparison

Consider a helper function that compares two objects of the same class. The helper returns true if their data values are identical and false otherwise.

Example

Let us add two queries ***(getNo()*** and ***getGrades()***) to our class definition and introduce a free helper function named ***areIdentical()*** to support our ***Student*** class. We insert the prototype for our helper function into the header file *after* the class definition:

```
// Student.h

 const int M = 13;

 class Student {
     int no;
     char grade[M+1];
 public:
     Student();
     Student(int);
     Student(int, const char*);
     void display() const;
     const Student& operator+=(char);
     int getNo() const { return no; }
     const char* getGrades() const { return grade; }
 };

 bool areIdentical(const Student&, const Student&);
```

The implementation file contains the definition of our helper function.

```
// Student.cpp

 #include <iostream>
 #include <cstring>
 using namespace std;
 #include "Student.h"

 Student::Student() {
     no = 0;
     grade[0] = '\0';
 }

 Student::Student(int n) {
     *this = Student(n, "");
```

```
}

Student::Student(int n, const char* g) {
    // see Current Object chapter for validation logic
    no = n;
    strcpy(grade, g);
}

void Student::display() const {
    cout << no << ' ' << grade;
}

const Student& Student::operator+=(char g) {
    int i = strlen(grade);
    if (i < M) {
        // there is room to add g
        grade[i++] = g;
        grade[i] = '\0';
    }
    return *this;
}

bool areIdentical(const Student& lhs, const Student& rhs) {
    return lhs.getNo() == rhs.getNo() &&
           strcmp(lhs.getGrades(), rhs.getGrades()) == 0;
}
```

The following client compares the two objects:

```
// Compare Objects
 // compare.cpp

#include <iostream>
using namespace std;
#include "Student.h"

int main () {
    Student harry(975,"AAAAA"), josee(975,"AAAAA");

    if (areIdentical(harry, josee))
        cout << "are identical" << endl;
    else
        cout << "are different" << endl;
}
```

```
are identical
```

The Cost of Freedom

Free helper functions require queries for information that is otherwise not accessible through existing public member functions. If we add a data member to the class, we may also need to add a query to access its value. The class definition grows with the addition of each new query. We call this growth *class bloat*.

To offset class bloat, we introduce friendship (as described below).

Helper Operators

Helper operators are overloaded operators that are global functions. Candidates for helper operators are those that do not change the values of thier operands as shown in the table below.

Effect on Operand(s)	Candidate	Operands	Operator
Left Operand Changes	Member	0	++ -- - + ! & *
		1	= += -= *= /= %=
Neither Operand Changes	Helper	2	+ - * / % == != >= <= > < << >>

Comparison

To improve readability, let us replace our areIdentical() function with an overloaded == operator that also takes two Student operands. The header file for the Student class now contains:

```
// Student.h

 const int M = 13;

 class Student {
     int no;
     char grade[M+1];
 public:
     Student();
     Student(int);
     Student(int, const char*);
     void display() const;
     const Student& operator+=(char);
     int getNo() const { return no; }
     const char* getGrades() const { return grade; }
 };

 bool operator==(const Student&, const Student&);
```

This helper operator accesses the private data of each operand through queries:

```
bool operator==(const Student& lhs, const Student& rhs) {
    return lhs.getNo() == rhs.getNo() &&
           strcmp(lhs.getGrades(), rhs.getGrades()) == 0;
 }
```

Addition

Let us overload the + operator to add a single grade to a Student object and return a copy. The left left operand is a Student and the right operand is a char. The header file for the Student class now contains:

```
// Student.h

 const int M = 13;

 class Student {
     int no;
     char grade[M+1];
 public:
     Student();
     Student(int);
     Student(int, const char*);
     void display() const;
     const Student& operator+=(char);
     int getNo() const { return no; }
     const char* getGrades() const { return grade; }
 };
 bool operator==(const Student&, const Student&);
 Student operator+(const Student&, char);
```

Our implementation avoids accessing private data by initializing a new Student object to the left operand and calling the += member operator on that object to add the right operand:

```
Student operator+(const Student& s, char grade) {
     Student copy = s; // makes a copy
     copy += grade;    // calls the += operator on copy
     return copy;      // return updated copy
 }
```

For symmetry, we overload this operator for identical operand types in reverse order. The complete header file contains:

```
// Student.h

 const int M = 13;

 class Student {
     int no;
     char grade[M+1];
 public:
     Student();
     Student(int);
     Student(int, const char*);
     void display() const;
     const Student& operator+=(char);
     int getNo() const { return no; }
     const char* getGrades() const { return grade; }
 };

 bool operator==(const Student&, const Student&);
```

```
Student operator+(const Student&, char);
Student operator+(char, const Student&);
```

Our implementation calls the original version with the arguments switched:

```
// Student.cpp

#include <iostream>
#include <iomanip>
using namespace std;
#include "Student.h"

Student::Student() {
    no = 0;
    grade[0] = '\0';
}

Student::Student(int n) {
    *this = Student(n, "");
}

Student::Student(int n, const char* g) {
    // see Current Object chapter for validation logic
    no = n;
    strcpy(grade, g);
}

void Student::display() const {
    cout << no << ' ' << grade;
}

const Student& Student::operator+=(char g) {
    int i = strlen(grade);
    if (i < M) {
        // there is room to add g
        grade[i++] = g;
        grade[i] = '\0';
    }
    return *this;
}

bool operator==(const Student& lhs, const Student& rhs) {
    return lhs.getNo() == rhs.getNo() &&
           strcmp(lhs.getGrades(), rhs.getGrades()) == 0;
}

Student operator+(const Student& student, char grade) {
    Student copy = student; // makes a copy
    copy += grade;          // calls the += operator on copy
    return copy;            // return updated copy
}

Student operator+(char grade, const Student& student) {
    return student + grade; // calls operator+(const
                            //    Student&, char)
}
```

The following client code produces the results shown on the right:

```
// Helper Operator
 // helper-operator.cpp

 #include <iostream>
 using namespace std;
 #include "Student.h"

 int main () {
     Student harry(975,"AAAAA");

     harry.display();
     cout << endl;
     harry = harry + 'B';
     harry.display();
     cout << endl;
 }
```

```
 975 AAAAA

 975 AAAAAB
```

Friendship

Friendship grants access to all private members of a class. By granting a helper function friendship status, a class allows that helper function access to any of its private members: data members or member functions. Friendly helper functions minimize class bloat.

To grant a helper function friendship status, we declare the function a *friend*. A friendship declaration takes the form

```
friend Type identifier(...);
```

where ***Type*** is the return type of the function and ***identifier*** is the function's name.

For example:

```
// Student.h

 const int M = 13;

 class Student {
     int no;
     char grade[M+1];
 public:
     Student();
     Student(int);
     Student(int, const char*);
     void display() const;
     const Student& operator+=(char);
     friend bool operator==(const Student&, const Student&);
 };

 Student operator+(const Student&, char);
 Student operator+(char, const Student&);
```

Our implementation looks like:

```
// Student.cpp

#include <iostream>
#include <cstring>
using namespace std;
#include "Student.h"

Student::Student() {
    no = 0;
    grade[0] = '\0';
}

Student::Student(int n) {
    *this = Student(n, "");
}

Student::Student(int n, const char* g) {
    // see Current Object chapter for validation logic
    no = n;
    strcpy(grade, g);
}

void Student::display() const {
    cout << no << ' ' << grade;
}

const Student& Student::operator+=(char g) {
    int i = strlen(grade);
    if (i < M) {
        // there is room to add g
        grade[i++] = g;
        grade[i] = '\0';
    }
    return *this;
}

bool operator==(const Student& lhs, const Student& rhs) {
    return lhs.no == rhs.no &&
           strcmp(lhs.grade, rhs.grade) == 0;
}

Student operator+(const Student &s, char grade) {
    Student copy = s; // makes a copy
    copy += grade;    // calls the += operator on copy
    return copy;      // return updated copy
}

Student operator+(char grade, const Student &s) {
    return s + grade; // calls operator+(const
                      //    Student&, char)
}
```

We have added the keyword friend only to the declaration within the class definition. We did not apply the keyword to our implementation.

The following client code that uses this implementation produces the results shown on the right:

```
// Friends
 // friends.cpp

 #include <iostream>
 using namespace std;
 #include "Student.h"

 int main () {
     Student harry(975,"AAAAA"), backup = harry;

     harry.display();                                          975 AAAAA
     cout << endl;
     harry = harry + 'B';
     harry.display();                                          975 AAAAAB
     cout << endl;
     if (harry == backup)
         cout << "identical" << endl;
     else
         cout << "different" << endl;                          different
 }
```

The Cost of Friendship

Friendship is the strongest relationship that a class can have with an outside entity. Friendship is not exclusive. A friend of one class may be a friend of any other class.

A class definition that grants friendship to a helper function allos that function to alter the values of its private data members. Granting friendship pierces encapsulation.

As a rule, we grant friendship judiciously only to helper functions that require both *read and write* access to the private data members. Where read-only access is all that a helper function needs, using queries is probably more advisable.

Friendly Classes

A class can grant access to its private members to all of the member functions of another class. A class friendship declaration takes the form

```
friend class Identifier;
```

where ***Identifier*** is the name of the class to which the host class grants friendship privileges.

For example, an Administrator class needs access to all of the information held within each Student object. To grant such access, we simply include a class friendship declaration within the Student class definition

```
// Student.h

 const int M = 13;

 class Student {
     int no;
     char grade[M+1];
 public:
     Student();
     Student(int);
     Student(int, const char*);
     void display() const;
     const Student& operator+=(char);
     friend bool areIdentical(const Student&, const Student&);
     friend class Administrator;
 };
```

No Reciprocity or Transitivity

Friendship is neither reciprocal nor transitive. Just because one class is a friend of another class does not mean that the latter is a friend of the former. Just because a class is a friend of another class and that other class is a friend of yet another class does not mean that the latter class is a friend of either of them.

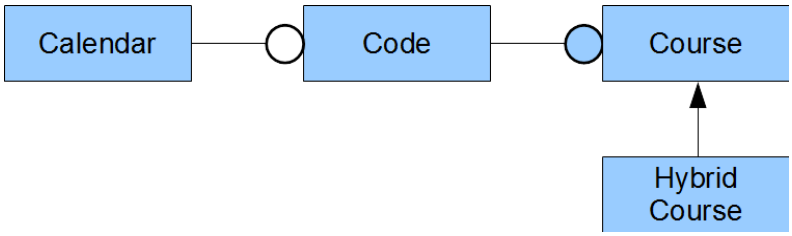
Consider three classes: a ***Student***, an ***Administrator*** and an ***Auditor***.

- Let the ***Auditor*** be a friend of the ***Administrator*** and the ***Administrator*** be a friend of the ***Student***
- Just because the ***Auditor*** is a friend of any ***Administrator*** and the ***Administrator*** is a friend of any ***Student***, the ***Administrator*** is not necessarily a friend of the ***Auditor*** and the ***Student*** is not necessarily a friend of the ***Administrator*** (lack of reciprocity)
- Just because the ***Auditor*** is a friend of any ***Administrator*** and the ***Administrator*** is a friend of any ***Student***, the ***Auditor*** is not necessarily a friend of any ***Student*** (lack of transitivity)

1.8 Custom I/O Operators

Object-oriented languages support a variety of relationships between classes. These include compositions and associations, both of which are looser than friendship. A *composition* is a relationship in which one class *has* another class. An *association* is a relationship in which one class *uses* another class. Associations are looser than compositions.

Relationship diagrams identify associations using open circles and compositions using filled circles. The diagram below shows that Code uses the Calendar (to determine availability), while each Course has a Code class.



To associate our own classes with those of the ***iostream*** library stream classes we overloading the insertion and extraction operators as helper operators that take ***iostream*** objects as left operands and objects of our class type as right operands.

This chapter describes how to overload the insertion and extraction operators for objects of our own class type. Thechapter concludes by introducing the standard library's ***string*** class, which is quite useful in managing character stream input of user-defined length.

Design Considerations

The C++ operators for inserting values into an output stream and extracting values from an input stream are:

- << (insert into an output stream)
- >> (extract from an input stream)

The ***iostream*** library overloads these operators for ***std::ostream/std::istream*** objects as left operands and fundamental types as right operands. The library also defines the standard output and input objects (***cout, cin***).

We adopt scope resolution notation to refer to these entities each of which is defined in the standard namespace ***(std::):***

```
#include <iostream>

 int main() {
     int x;

     std::cout << "Enter an integer : ";
     std::cin >> x;
     std::cout << "You entered " << x <<
std::endl;
 }
```

```
 Enter an integer :
3

 You entered 3
```

Objective

We wish to associate our ***Student*** class with the ***iostream*** classes, so that any client can use the insertion and extraction operators to process objects of our class:

```
#include <iostream>
 #include "Student.h"

 int main() {
     Student harry;

     std::cin >> harry;
     std::cout << harry << std::endl;
 }
```

```
 Enter number : 1234
 Enter grades : ABACA
 1234 ABACA
```

Good Design

In overloading the insertion and extraction operators for our class types, we intend to:

- provide flexibility in the selection of output objects
- use scope resolution on classes and objects defined in the std namespace
- enable cascading as implemented for fundamental types

Selection of Output Objects

To enable selection of the output object, we upgrade our ***display()*** member function to receive a reference to an object of ***std::ostream*** type:

```
// Student.h

 #include <iostream> // for
std::ostream

 const int M = 13;

 class Student {
     int no;
     char grade[M+1];
 public:
     Student();
     Student(int);
     Student(int, const char*);
     void display(std::ostream& os)
      const;
 };
```

```
 // Student.cpp

 #include <cstring>
 #include "Student.h"

 Student::Student() {
     no = 0;
     grade[0] = '\0';
 }
 // ...

 void
Student::display(std::ostream&
  os) const {
     os << no << ' ' << grade;
 }
 // ...
```

With this code the client chooses the output object (cout, cerr, clog).

Scope Resolution

Scope resolution on the ***display()*** function's parameter exposes the class that we actually use, without exposing any other name defined in the std namespace.

This convention is important in coding header files. Exposing all of the names in any namspace may lead to unnecessary conflicts with new names or conflicts when several header files are included in an implementation file.

The preferred method of coding header files is shown on the right:

```
// Student.h

#include <iostream>
using namespace std; // POOR DESIGN
const int M = 13;

class Student {
    int no;
    char grade[M+1];
public:
    Student();
    Student(int);
    Student(int, const char*);
    void display(ostream& os) const;
};
```

```
// Student.h

#include <iostream>  // GOOD DESIGN
const int M = 13;

class Student {
    int no;
    char grade[M+1];
public:
    Student();
    Student(int);
    Student(int, const char*);
    void display(std::ostream& os)
     const;
};
```

Cascading

Cascading support enables concatenation of operations where the leftmost operand serves as the left operand for every operation in a compound expression.

For example, the cascaded expression

```
std::cout << x << y << z << std::endl;
```

expands to two simpler sub-expressions executed in the following order:

```
std::cout << x;
std::cout << y << z << std::endl;
```

The cascaded sub-expression

```
std::cout << y << z << std::endl;
```

expands to two simpler sub-expressions executed in the following order:

```
std::cout << y;
std::cout << z << std::endl;
```

Finally, the cascaded sub-expression

```
std::cout << z << std::endl;
```

expands into two simpler sub-expressions executed in the following order:

```
std::cout << z;
std::cout << std::endl;
```

To enable cascading, we return a modifiable reference to the left operand.

Returning a modifiable reference from a function enables the client to use the return value as the left operand for the operator on its right. The call to an operator that returns a modifiable reference takes the following form after returning from the function:

```
return value   next operator   next right operand
```

The next right operand may be a compound expression with more operators as shown in the above example.

Two Helper Operators

The prototypes for overloading insertion and extraction operators on our own classes take the form

```
std::istream& operator>>(std::istream&, Type&);
std::ostream& operator<<(std::ostream&, const Type&);
```

where ***Type*** is the name of the class.

The header file for our ***Student*** class upgraded for these custom operators is:

```
// Student.h

#include <iostream> // for std::ostream, std::istream
const int M = 13;

class Student {
    int no;
    char grade[M+1];
public:
    Student();
    Student(int);
    Student(int, const char*);
    void display(std::ostream& os) const;
    operator bool() const { return no == 0; }
};

std::istream& operator>>(std::istream& is, Student& s);
std::ostream& operator<<(std::ostream& os, const Student& s);
```

The ***bool*** conversion operator lets the client know if the object is in a safe empty state.

The implementation file for our upgraded ***Student*** class contains:

```
// Student.cpp

#include <cstring>
#include "Student.h"

Student::Student() {
    no = 0;
    grade[0] = '\0';
}

Student::Student(int n) {
    *this = Student(n, "");
}

Student::Student(int n, const char* g) {
    // see Current Object chapter for validation logic
    no = n;
    std::strcpy(grade, g);
}

void Student::display(std::ostream& os) const {
    os << no << ' ' << grade;
}

std::ostream& operator<<(std::ostream& os, const Student& st){
    st.display(os);
    return os;
}

std::istream& operator>>(std::istream& is, Student& s) {
```

```
    int no;
    char grade[M+1];

    // student number
    std::cout << "Number : ";
    is >> no;

    // student grades
    std::cout << "Grades : ";
    is.ignore();             // swallow newline in the buffer
    is.getline(grade, M+1);  // read string with whitespace

    Student temp(no, grade);
    if (!temp)
        s = temp;  // replace s only if not empty
    return is;
}
```

The ***getline()*** member function receives in its second parameter the size of the C-style string that accepts the input data, which including room for the null byte terminator.

The following client uses our upgraded Student class to produce the results shown on the right:

```
// Custom I/O Operators
 // customIO.cpp

 #include "Student.h"

 int main () {
     Student harry;

     std::cin >> harry;
     std::cout << harry << std::endl;
 }
```

```
Number : 1234
Grades : ABACA
1234 ABACA
```

In this solution we assume that the user does not enter characters that would cause a stream failure. To process failures, we include validation logic that tests the state of the input object and requests corrected input as required (see the Robust Validation section in the chapter on Input and Output Examples).

String Class

The Problem

The solution in the above example can fail if the user enters more than M grades. Determining the amount of memory needed to hold the grades that the user chooses to input requires a dynamic solution. Since we do not know how much memory to allocate before receiving all of the user's input, we cannot predict at compile time the maximum size of memory that any client will require.

The Solution

The ***string*** class of the standard library provides a solution that allocates the required amount of memory at run-time. A ***string*** object can accept as many characters as the user enters. The helper function ***getline()*** extracts them from the input stream.

The prototype for this helper function is

```
std::istream& getline(std::istream&, std::string&, char);
```

The first parameter receives a modifiable reference to the ***istream*** object, the second parameter receives a modifiable reference to the ***string*** object and the third parameter receives the character delimiter for terminating the extraction (newline by default).

The <***string***> header file contains the class definition with this prototype. The class definition includes two member functions for converting its internal data into a C-style null-terminated string:

- ***std::string::length()*** - returns the number of characters in the string
- ***std::string::c_str()*** - returns the address of the C-style null-terminated version of the string

Preliminary Example

The following client extracts an unknown number of characters from the standard input stream and displays the characters on the standard output object. The extraction involves five steps:

- define a string object to accept the input
- extract the input using the getline() helper function
- query the memory required for a C-style null terminated string
- allocate dynamic memory for the C-style null-terminated string
- copy the input data from the string object into the allocated memory

```
// String class example
 // string.cpp

 #include <iostream>
 #include <string>

 int main( ) {
     char* s;
     std::string str;

     std::cout << "Enter a string : ";
     if (std::getline(std::cin, str)) {
```

```
        s = new char [str.length() + 1];
        std::strcpy(s, str.c_str());
        std::cout << "The string entered is : >" << s << '<' << std::endl;
        delete [] s;
    }
}
```

Student Class Example

To upgrade our ***Student*** class to accept any number of characters, we use a local ***string*** object in our overload of the extraction operator.

The header file for our ***Student*** class contains:

```
// Student.h

#include <iostream>

class Student {
    int no;
    char* grade;
public:
    Student();
    Student(int, const char*);
    Student(const Student&);
    Student& operator=(const Student&);
    ~Student();
    operator bool() const { return no == 0; }
    void display(std::ostream&) const;
};

std::istream& operator>>(std::istream& is, Student& s);
std::ostream& operator<<(std::ostream& os, const Student& s);
```

The implementation file contains:

```
// Student.cpp

#include <cstring>
#include <string>
#include "Student.h"

Student::Student() {
   // safe empty state
   no = 0;
   grade = nullptr;
}

Student::Student(int n) {
   grade = nullptr;
   *this = Student(n, "");
}
```

```
Student::Student(int n, const char* g) {
    int i;
    bool valid = true; // assume valid input, check invalidity

    // validate client data
    if (n < 1)
        valid = false;
    else {
        for (i = 0; g[i] != '\0' && valid; i++)
            valid = g[i] >= 'A' && g[i] <= 'F' && g[i] != 'E';

    if (valid) {
        // accept client data
        no = n;
        grade = new char[std::strlen(g) + 1];
        std::strcpy(grade, g);
    }
    else {
        // set to a safe empty state
        grade = nullptr;
        *this = Student();
    }
}

Student::~Student() {
    delete [] grade;
}

void Student::display(std::ostream& os) const {
    os << no << ' '
       << ((grade != nullptr) ? grade : "");
}

std::ostream& operator<<(std::ostream& os, const Student& s) {
    s.display(os);
    return os;
}

std::istream& operator>>(std::istream& is, Student& s) {
    bool ok;
    int number;
    std::string grade;

    // student number
    std::cout << "Number : ";
    is >> number;

    // student grades
    std::cout << "Grades : ";
    is.ignore();
    if (std::getline(is, grade)) {
        Student temp(number, grade.c_str());
        if (!temp)
            s = temp;
    }
    else {
        is.clear();
```

```
        is.ignore(2000, '\n');
    }
    return is;
}
```

The extraction operator only stores the input in the right operand if the getline() function successfully reads the grade input.

```
// String Class
 // string.cpp

 #include <iostream>
 #include "Student.h"

 int main ( ) {
    Student harry;

    std::cin >> harry;
    std::cout << harry << std::endl;
 }
```

```
Number : 1234
Grades : AABBCAADFBBCABBA
1234 AABBCAADFBBCABBA
```

1.9 Custom File Operators

File stream classes share much of their structure with the standard input and output classes described in the chapter entitled Input and Output Examples. The ***iostream*** library associates its file objects with the fundamental types by overloading the extraction and insertion operators for those types. We can overload these operators for objects of our own class type as right operands in the way that we did in the preceding chapter for the standard input and output objects as left operands.

This chapter introduces the stream classes that manage communication with file objects. The chapter describes how to create file objects from these classes, how to read and write data of fundamental type and how to overload the operators for file objects as left operands and objects of our own class type as right operands.

File Stream Classes

The ***iostream*** library defines three classes for managing communication between file streams containing 8-bit characters and system memory:

- ***ifstream*** - processes input from a file stream
- ***ofstream*** - processes output to a file stream
- ***fstream*** - processes input from and output to a file stream

These classes provide access to a file stream through separate input and output buffers.

The ***fstream*** system header file defines these classes in the ***std*** namespace:

```
#include <fstream>
```

State Methods

The queries for interrogating a file object include:

- ***bool is_open() const*** **-** the file stream is open
- ***bool good() const*** **-** the file stream is ready to process
- ***bool fail() const*** **-** the file object has failed
- ***bool eof() const*** **-** the file object has encountered an end of file mark
- ***bool bad() const*** **-** the file object has encountered a serious error

If a file object is not in a good() state, we must reset its state. To reset state we call the modifier:

- ***void clear()*** **-** resets the file object to a good state

These member functions operate in the same way as described in the chapter entitled Input and Output Examples for the standard input and output objects.

File Objects

A file object is an instance of one of the file stream classes. When used with the insertion or extraction operators, a file object processes data in formatted form. The object uses the host platform's encoding sequence (ASCII, EBCDIC, Unicode) in converting from stream bytes to data stored in system memory and vice versa.

File Connection

We can connect a file object to a file for reading, writing or both. The object's destructor closes the file connection.

Input Objects

We create a file object for reading by defining an instance of the ***ifstream*** class. This class defines a no-argument constructor and one that receives the address of a C-style null-terminated string holding the name of the file.

For example,

```
#include <fstream>
```

```
 std::ifstream fin("input.txt"); // connects fin to input.txt for reading
```

To connect a file to an existing file object, we call the open() member function on the object.

For example,

```
#include <fstream>

 std::ifstream fin;     // defines a file object named fin
 fin.open("input.txt"); // connects input.txt to fin
```

Output Objects

We create a file object for writing by defining an instance of the ofstream class. This class defines a no-argument constructor and one that receives the address of a C-style null-terminated string holding the name of the file.

For example,

```
#include <fstream>

 std::ofstream fout("output.txt"); // connects fout to output.txt for
writing
```

To connect a file to an existing file object, we call the open() member function on the object.

For example,

```
#include <fstream>

 std::ofstream fout;      // create a file object named fout
 fout.open("output.txt"); // connects output.txt to fout
```

Confirming the Connection

The ***is_open()*** member function on a file object returns the current state of the connection:

```
#include <iostream>
 #include <fstream>

 std::ofstream fout("output.txt");  // connects output.txt to fout for
output

 if (!fout.is_open()) {
     std::cerr << "File is not open" << std::endl;
 } else {
     // file is open

 }
```

Fundamental Types

We use the same syntax to read from a file and write to a file as we use to read from the standard input object and write to a standard output object (see the chapter entitled Input and Output Examples for a review).

The standard library contains overloads of the extraction and insertion operators for each fundamental type with objects of the file stream classes as left operands.

Reading From a File

A file object reads from a file under format control using the extraction operator in the same way as the standard input object (cin) does using the operator.

For example, consider a file with a single record: 12 34 45 abc The output from the following program is shown on the right:

```
// Reading a File
 // readFile.cpp

 #include <iostream>
 #include <fstream>

 int main() {
     int i;

     std::ifstream f("input.txt");
     if (f.is_open()) {
         while (f.good()) {
             f >> i;
             if (f.good())
                 std::cout << i << ' ';
             else if (!f.eof())
                 std::cout << "\n**Bad input**\n";
         }
     }
 }
```

```
 12 34 45

 **Bad input**
```

Writing to a File

A file object writes to its connection under format control using the insertion operator in the same way as the standard output objects (***cout***, ***cerr*** and ***clog***) do using the operator.
For example, the contents of the file created by the following program are shown on the right

```
// Writing to a File
 // writeFile.cpp

 #include <iostream>
 #include <fstream>

 int main() {
     int i;

     std::ofstream f("output.txt");
     if (f.is_open()) {
         f << "Line 1" << std::endl;   // record 1
         f << "Line 2" << std::endl;   // record 2
         f << "Line 3" << std::endl;   // record 3
     }
 }
```

```
Line 1
Line 2
Line 3
```

Custom Types

Insertion and extraction operators that have been overloaded for standard output and input objects respectively as left operands and a custom type as the right operand work without modification with file objects as left operands. This flexibility has to do with inheritance, which is described later in the chapter entitled Functions in a Hierarchy. Neither the header file nor the implementation file require any modification.
Since accepting input from a file does not involve the interaction that we expect across a standard input device, we typically overload the extraction operator to work differently with file objects. We overload the operator for an ifstream object as the left operand. Our overload of the ostream operator holds for output to a file stream as well as to the standard output stream.
For example, we add the prototype for the file extraction helper to the definition of our Student class:

```
// Student.h

 #include <iostream> // for std::ostream, std::istream
 #include <fstream>  // for std::ifstream
 const int M = 13;
```

```
class Student {
    int no;
    char grade[M+1];
public:
    Student();
    Student(int);
    Student(int, const char*);
    Student& operator+=(char);
    void display(std::ostream& os) const;
    operator bool() const { return no == 0; }
};

std::istream& operator>>(std::istream& is, Student& s);
std::ifstream& operator>>(std::ifstream& is, Student& s);
std::ostream& operator<<(std::ostream& os, const Student& s);
```

The implementation file contains the definition of the file extraction operation for our ***Student*** class:

```
// Student.cpp

#include <cstring>
#include "Student.h"

Student::Student() {
    no = 0;
    grade[0] = '\0';
}

Student::Student(int n) {
    *this = Student(n, "");
}

Student::Student(int n, const char* g) {
    // see Current Object chapter for validation logic
    no = n;
    std::strcpy(grade, g);
}

void Student::display(std::ostream& os) const {
    os << no << ' ' << grade;
}

Student& Student::operator+=(char g) {
    int i = strlen(grade);
    if (i < M) {
        // add validation logic here
        grade[i++] = g;
        grade[i] = '\0';
    }
    return *this;
}

std::ostream& operator<<(std::ostream& os, const Student& st) {
```

```
    st.display(os);
    return os;
}

std::istream& operator>>(std::istream& is, Student& s) {
    int no;
    char grade[M+1];

    // student number
    std::cout << "Number : ";
    is >> no;

    // student grades
    std::cout << "Grades : ";
    is.ignore();             // swallow newline in the buffer
    is.getline(grade, M+1);  // read string with whitespace

    Student temp(no, grade);
    if (!temp)
        s = temp;  // replace s only if not empty
    return is;
}

std::ifstream& operator>>(std::ifstream& is, Student& s) {
    int no;
    char grade[M+1];

    // student number
    is >> no;
    is.ignore(); // skip whitespace

    // student grades
    is.getline(grade, M+1);  // read string with whitespace

    Student temp(no, grade);
    if (!temp)
        s = temp;  // replace s only if not empty
    return is;
}
```

Compare the overloaded definitions for the extraction operator. The ***ifstream*** definition omits user prompts.

The client file that uses this upgraded ***Student*** class creates the file objects, writes to them and reads from them:

```
// Custom File Operators
 // customFile.cpp

#include "Student.h"

int main ( ) {
    Student harry(975, "AABD"), josee(976, "BAAA");

    std::ofstream oufile("Student.txt");
    oufile << harry << std::endl;
```

```
    oufile << josee << std::endl;
    oufile.close();
    std::cout << harry << std::endl;                    975 AABD
    std::cout << josee << std::endl;                    976 BAAA

    std::ifstream infile("Student.txt");
    infile >> harry;
    infile >> josee;
    harry += 'B';
    josee += 'C';
    std::cout << harry << std::endl;                    975 AABDB
    std::cout << josee << std::endl;                    976 BAAAC
}
```

The records written to the ***Student.txt*** file by this program are:

```
 975 AABD
 976 BAAA
```

Nice To Know
Open-Mode Flags

We customize a file object's connection mode through combinations of flags passed as an optional second argument to the object's constructor or its ***open()*** member function.

The flags defining the connection mode are:

- ***std::ios::in*** open for reading
- ***std::ios::out*** open for writing
- ***std::ios::app*** open for appending
- ***std::ios::trunc*** open for writing, but truncate if file exists
- ***std::ios::ate*** move to the end of the file once the file is opened

Practical combinations of these flags include

- ***std::ios::in*|*std::ios::out*** open for reading and writing (default)
- ***std::ios::in*|*std::ios::out*|*std::ios::trunc*** open for reading and overwriting
- ***std::ios::in*|*std::ios::out*|*std::ios::app*** open for reading and appending
- ***std::ios::out*|*std::ios::trunc*** open for overwriting

The vertical bar **(|)** is the *bit-wise or* operator.

The Defaults

The default combinations of the no-argument and one-argument constructors are:

- ***ifstream - std::ios::in*** - open for reading
- ***ofstream - std::ios::out*** - open for writing
- **fstream - std::ios::in|std::ios::out** - open for reading and writing

The Logical Negation Operator

The standard library overloads the logical negation operator **(!)** as an alternative to the **fail()** query. This operator reports ***true*** if the latest operation has failed or if the stream has encountered a serious error.

We can invoke this operator on any stream object to check the success of the most recent activity:

```
if (fin.fail()) {
     std::cerr << "Read error";
     fin.clear();
 }
```

```
 if (!fin) {
     std::cerr << "Read error";
     fin.clear();
 }
```

The operator applied directly to a file object returns the state of the connection:

```
#include <iostream>
#include <fstream>

 std::ofstream fout("output.txt");  // connects fout to output.txt for
writing

 if (!fout) {
     std::cerr << "File is not open" << std::endl;
 } else {
     // file is open

 }
```

Rewinding a Connection
istream, fstream

To rewind an input stream we call:

- ***istream& seekg(0)*** - sets the current position in the input stream to **0**

ostream, fstream

To rewind an output stream we call:

- ***ostream& seekp(0)*** - sets the current position in the output stream to **0**

Premature Closing

To close a file connection before the file object has gone out of scope, we call the close() member function on the object:

```
// Concatenate Two Files
 // concatenate.cpp
```

```
#include <fstream>

int main() {
    std::ifstream in("src1.txt");    // open 1st source file
    std::ofstream out("output.txt"); // open destination file

    while (!in.eof())
        out << in.get();         // byte by byte copy
    in.clear();
    in.close();                  // close 1st source file
    in.open("src2.txt");         // open 2nd source file

    while (!in.eof())
        out << in.get();         // byte by byte copy
    in.clear();
}
```

Writing to and Reading from the Same File

An instance of the ***fstream*** class can write to a file and read from that same file.

For example, the following program produces the output shown on the right

```
// File Objects - writing and reading
 // fstream.cpp

#include <iostream>
#include <fstream>

int main() {

    std::fstream f("file.txt",
     std::ios::in|std::ios::out|std::ios::trunc);
    f << "Line 1" << std::endl;   // record 1
    f << "Line 2" << std::endl;   // record 2
    f << "Line 3" << std::endl;   // record 3
    f.seekp(0);                   // rewind output
    f << "****";                  // overwrite

    char c;
    f.seekg(0);                   // rewind input
    f << std::noskipws;           // don't skip whitespace
    while (f.good()) {
        f >> c;                   // read 1 char at a time
        if (f.good())
            std::cout << c;       // display the character
    }
    f.clear();                    // clear failed (eof) state
}
```

```
**** 1
Line 2
Line 3
```

1.10 Check Your Progress

Q.1 What is the difference between Constructor and Destructor? Explain with the help of an example.

Q.2 What is Default Constructor and Default Destructor? Give an example.

Q.3 What is Copy Constructor? Give example.

Q.4 What is Assignment Operator?

Q.5 What is Binary and Unary Operations?

Q.6 What are Custom I/O Operators?

Q.7 What are Custom File Operators?

Block-4

1.1 Learning Objectives

After going through this unit, the learner will able to:

- Understand the concept of Inheritance
- Define derived classes
- Learn about the functions hierarchy
- Learn and define derived classes and resources

1.2 Introduction to Inheritance

In object-oriented programming, **inheritance** is when an object or class is based on another object (prototypal inheritance) or class (class-based inheritance), using the same implementation (inheriting from an object or class) specifying implementation to maintain the same behavior (realizing an interface; inheriting behavior). It is a mechanism for code reuse and to allow independent extensions of the original software via public classes and interfaces. The relationships of objects or classes through inheritance give rise to a hierarchy. Inheritance was invented in 1967 for Simula. The term "inheritance" is loosely used for both class-based and prototype-based programming, but in narrow use is reserved for class-based programming (one class *inherits from* another), with the corresponding technique in prototype-based programming being instead called *delegation* (one object *delegates to* another).

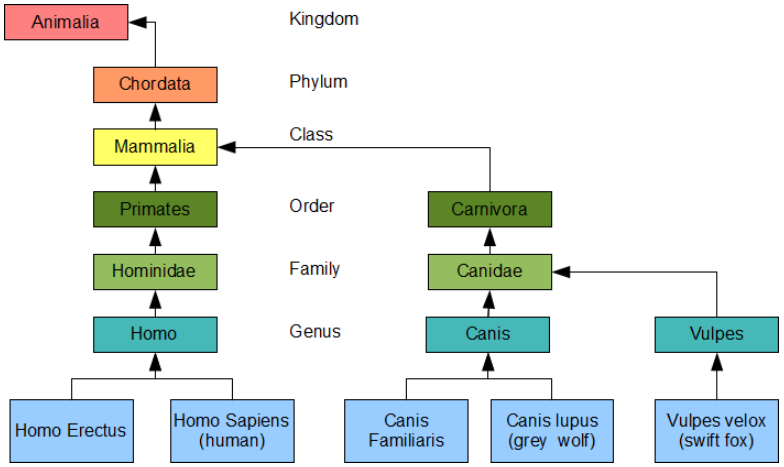
1.3 Derived Classes

Inheritance is the second prominent concept that object-oriented languages implement and is closely related to encapsulation. *Inheritance* refers to the relationship between classes where one class inherits the entire structure of another class. Inheritance is naturally hierarchical, a tighter relationship than composition and the most highly coupled after friendship.

This chapter introduces hierarchy, the terminology used to describe inheritance and the syntax for defining a class that inherits the structure of another class. This chapter also covers accessibility privileges between classes within a hierarchy.

Hierarchies

A comprehensive example of inheritance relationships is the Linnean Hierarchy in Biology (a small portion is shown below). This hierarchy relates all biological species in existence to one another. Starting from the bottom of the hierarchy, we identify a human as a homo, which is a hominidae, which is a primate, which is a mammal, which is a chordata, which is an animal. Similarily a dog is a canis, which is a canidae, which is a carnivora, which is a mammal, which is a chordata, which is an animal.



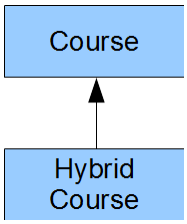
Carl Linnaeus earned himself the title of Father of Taxonomy after developing this hierarchy. He grouped the genera of Biology into higher taxa based on shared similarities. Using his taxa with its modern refinements, we say that the *genus* Homo, which includes the *species* sapiens, belongs to the *Family* Hominidae, which belongs to the *Order* Primates, which belongs to the *Class* Mammalia, which belongs to the *Phylum* Chordata, which belongs to the *Kingdom* Animalia. For more details see the University Of Michigan Museum Of Zoology's Animal Diversity Site.

Inheritance is a *transitive* relationship. A human inherits the structure of a homo, which inherits the structure of a hominoid, which inherits the structure of a primate, which inherits the structure of a mammal, which inherits the structure of a chordata, which inherits the structure of an animal.

Inheritance is *not commutative*. A primate is an animal, but an animal is not necessarily a primate: dogs and foxes are not primates. Primates have highly developed hands and feet, shorter snouts and larger brains than dogs and foxes.

Terminology

The relative position of two classes in a hierarchy identifies their inheritance relationship. A class lower in the hierarchy *is a kind of* the class that is higher in the hierarchy. For example, a dog *is a kind of* canis, a fox *is a kind of* vulpes and a human *is a kind of* homo. In our course example from the first chapter, a ***Hybrid Course*** *is a kind of* ***Course***. We depict inheritance by an arrow pointed towards the inherited class.

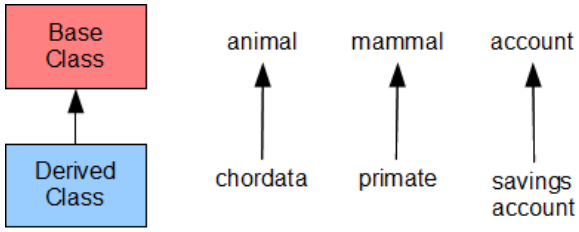


The ***Hybrid*** Course class inherits the entire structure of the Course class.

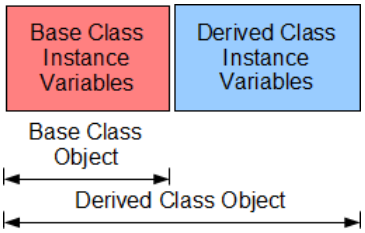
Base and Derived

We call the Course class the *base* class and the Hybrid Course class the *derived* class. A derived class is lower in the hierarchy, while its base class is higher in the hierarchy. The derived class inherits the entire structure of its base class. The derived class is the subject of '*is-a-kind-of*', while the base class is the object.

We depict an inheritance relationship using an arrow drawn from the derived class to the base class:



We depict an instance of a derived class by drawing the instance variables of the derived class in the direction of increasing addresses with respect to the instance variables of its base class:



A derived class object contains the instance variables of the base class and those of the derived class, while a base class object only contains the instance variables of the base class.

The terms base class and derived class are C++ specific. Equivalent terms for these object-oriented concepts include:

- base class - super class, parent class
- derived class - subclass, heir class, child class

Inherited Structure

A derived class contains all of the instance variables and all of the normal member functions of its base class. The derived class does not inherit these special functions: constructors, destructors and assignment operators. Normal member functions exclude these *special* member functions.

Definition of a Derived Class

The definition of a derived class takes the form

```
class Derived : access Base {
    // ...
};
```

where ***Derived*** is the name of the derived class and ***Base*** is the name of the base class. ***access*** identifies the access that member functions of the derived class have to the non-private members of the base class. The default access is ***private***. The most common access modifier is ***public***.

Consider a ***Student*** ss a kind of ***Person***. Every ***Person*** has a name. Accordingly, we derive our ***Student*** class from a ***Person*** class.

The header file for our ***Student*** class contains the definitions of the base and derived classes:

```
// Student.h

 #include <iostream>
 const int N = 30;
 const int M  = 13;

 class Person {                    // Base Class - start
     char person[N+1];
   public:
     void set(const char* n);
     void displayName(std::ostream&) const;
 };                                // Base Class - end

 class Student : public Person { // Derived Class - start
     int no;
```

```
    char grade[M+1];
  public:
    Student();
    Student(int, const char*);
    void display(std::ostream&) const;
};                              // Derived Class - end
```

The implementation file defines the member functions:

```
// Student.cpp

#include <cstring>
#include "Student.h"

void Person::set(const char* n) {
    std::strncpy(person, n, N);
    person[N] = '\0';
}

void Person::displayName(std::ostream& os) const {
    os << person << ' ';
}

Student::Student() {
    no = 0;
    grade[0] = '\0';
}

Student::Student(int n, const char* g) {
    // add validation logic here
    no = n;
    std::strcpy(grade, g);
}

void Student::display(std::ostream& os) const {
    os << no << ' ' << grade;
}
```

The following client uses this implementation to produce the results on the right:

```
// Derived Classes
// derived.cpp

#include <iostream>
#include "Student.h"

int main() {
    Student harry(975, "ABBAD");

    harry.set("Harry");           // inherited
    harry.displayName(std::cout); // inherited
    harry.display(std::cout);     // not inherited
    std::cout << std::endl;
}
```

```
Harry 975 ABBAD
```

The ***main()*** function refers to the ***Student*** type, without mention of the ***Person*** type. That is, the hierarchy itself is invisible to the client. We can upgrade the hierarchy without having to alter the client code in any way.

Access

A derived class can have different levels of access to the members of its base class. C++ supports three access modifiers:

- **private** - bars all access
- p**rotected** - limits access to derived classes only
- **public** - unlimited access

The member functions of our **Student** class cannot access the data member of the **Person** class, since that member is **private** to the base class. On the other hand, the **main()** function and the member functions of the **Student** class can access the two member functions of **Person**, since those functions are **public**.

Limiting Access to Derived Classes

The keyword **protected** limits access to members of the derived classes.

For example, let us limit access to **displayName()** to derived classes. Then, the **main()** function cannot call this member function and we must call it directly from **Student::display()**. The header file limits the access:

```
// Student.h

 #include <iostream>
 const int N = 30;
 const int M = 13;

 class Person {
     char person[N+1];
   public:
     void set(const char* n);
   protected:
     void displayName(std::ostream&) const;
 };

 class Student : public Person {
     int no;
     char grade[M+1];
   public:
     Student();
```

```
    Student(int, const char*);
    void display(std::ostream&) const;
};
```

Our implementation of Student::display() calls displayName() directly:

```
// Student.cpp

#include <cstring>
#include "Student.h"

void Person::set(const char* n) {
    std::strncpy(person, n, N); // validates length
    person[N] = '\0';
}

void Person::displayName(std::ostream& os) const {
    os << person << ' ';
}

Student::Student() {
    no = 0;
    grade[0] = '\0';
}

Student::Student(int n, const char* g) {
    // insert validation logic here
    no = n;
    std::strcpy(grade, g);
}

void Student::display(std::ostream& os) const {
    displayName(os);
    os << no << ' ' << grade;
}
```

We refer to ***displayName()*** directly without any scope resolution as if this function is a member of our ***Student*** class.

The following client produces the output shown on the right:

```
// Protected Access
 // protected.cpp

#include <iostream>
#include "Student.h"

int main() {
    Student harry(975, "ABBAD");

    harry.set("Harry");
    harry.display(std::cout);
    std::cout << std::endl;
}
```

```
Harry 975 ABBAD
```

Avoid Granting Protected Access to Data Members

Granting data members protected access introduces a security hole. If a derived class has protected access to any data member of its base class, any member function of the derived class can circumvent any validation procedure in the base class. If the base class in the above example granted us access to the **person** data member, we could change its contents from our **Student** class to a string of more than N characters, which would probably break our **Student** object.

1.4 Functions in a Hierarchy

Inheritance treats normal member functions and the special member functions that manage objects differently. A derived class inherits the normal member functions of its base class. A derived class' destructor automatically calls the base class' destructor and no additional coding is required. A derived class' default assignment operator automatically calls the base class' assignment operator and no additional coding is required. Constructors in a class hierarchy are different because they are involved in the creation of objects: each constructor creates part of the final object.

This chapter examines how member functions shadow one another in a hierarchy, describes the order in which the compiler calls constructors and destructors, shows how to define a derived class' constructor to access a specific base class constructor and finally, describes how to overload a helper operator for a derived class.

Shadowing

A member function of a derived class *shadows* the base class member function with the same name. The compiler binds a call to the member function defined in the derived class, if one exists.

We use scope resolution to access the base class version of a member function that the derived class version has shadowed. A call to a shadowed function takes the form

```
Base::identifier(arguments)
```

where ***Base*** identifies the class that defines the shadowed function.

Example

Consider the following hierarchy. The base and derived classes define separate versions of the display() member function. The Student class version shadows the Person class version for any object of Student type:

```
// Student.h

 #include <iostream>
 const int N = 30;
 const int M = 13;

 class Person {
     char person[N+1];
   public:
     void set(const char* n);
     void display(std::ostream&) const;
 };

 class Student : public Person {
     int no;
     char grade[M+1];
   public:
     Student();
     Student(int, const char*);
     void display(std::ostream&) const;
 };
```

We access the base class version using scope resolution:

```
// Student.cpp

 #include <cstring>
 #include "Student.h"

 void Person::set(const char* n) {
     std::strncpy(person, n, N);
     person[N] = '\0';
 }

 void Person::display(std::ostream& os) const {
     os << person << ' ';
 }

 Student::Student() {
     no = 0;
     grade[0] = '\0';
 }

 Student::Student(int n, const char* g) {
     // insert validation logic here
     no = n;
     std::strcpy(grade, g);
 }
```

```
void Student::display(std::ostream& os) const {
    Person::display(os);
    os << no << ' ' << grade;
}
```

The following client produces the output shown on the right:

```
// Shadowing
 // shadowing.cpp

#include <iostream>
#include "Student.h"

int main() {
    Person jane;
    Student harry(975, "ABBAD");

    harry.set("Harry");
    harry.display(std::cout);
    std::cout << std::endl;
    jane.set("Jane Doe");
    jane.display(std::cout);
    std::cout << std::endl;
}
```

```
Harry 975 ABBAD

Jane Doe
```

harry.display(std::cout) calls ***Student::display(),*** which in turn calls the shadowed ***Person::display(). jane.display()*** calls ***Person::display()*** directly. The derived version shadows the base version on ***harry***, while the base version on ***jane*** is unshadowed.

Design Tip

By calling ***Person::display()*** within ***Student::display(),*** we hide the hierarchy from the client. The ***main()*** function is hierarchy egnostic.

Exposing Overloaded Member Functions

C++ shadows member functions on their name and not their signature. To expose a member function in the base class other than the function with the same signature we insert a ***using*** declaration into the definition of the derived class. A ***using*** declaration takes the form

```
using Base::identifier;
```

where *Base* identifies the base class and *identifier* is the name of the shadowed function.

Example

Let us overload the ***display()*** member function in the Person class to take no arguments. We insert the ***using*** declaration in the definition of the derived class to expose the member function for objects of the derived class.

```
// Student.h

 #include <iostream>
 const int N = 30;
 const int M = 13;

 class Person {
     char person[N+1];
   public:
     void set(const char* n);
     void display(std::ostream&) const;
     void display() const;
 };

 class Student : public Person {
     int no;
     char grade[M+1];
   public:
     Student();
     Student(int, const char*);
     void display(std::ostream&) const;
     using Person::display;
 };
```

We define the overloaded display() function for the base class:

```
// Student.cpp

 #include <cstring>
 #include "Student.h"

 void Person::set(const char* n) {
     std::strncpy(person, n, N);
     person[N] = '\0';
 }

 void Person::display(std::ostream& os) const {
     os << person << ' ';
 }

 void Person::display() const {
     std::cout << person << ' ';
 }

 Student::Student() {
     no = 0;
```

```
    grade[0] = '\0';
}

Student::Student(int n, const char* g) {
    // insert validation logic here
    no = n;
    std::strcpy(grade, g);
}

void Student::display(std::ostream& os) const {
    Person::display(os);
    os << no << ' ' << grade;
}
```

The following client produces the result shown on the right:

```
// Overloading and Shadowing
// overloading.cpp

#include <iostream>
#include "Student.h"

int main() {
    Person jane;
    Student harry(975, "ABBAD");

    harry.set("Harry");
    harry.display(std::cout);
    std::cout << std::endl;
    harry.display();
    std::cout << std::endl;

    jane.set("Jane Doe");
    jane.display(std::cout);
    std::cout << std::endl;
}
```

```
Harry 975 ABBAD
Harry

Jane Doe
```

Constructors

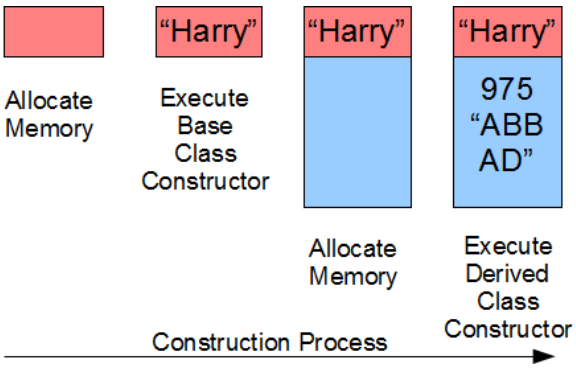
A derived class does not inherit a base class constructor by default. That is, if we don't declare a constructor in our definition of the derived class, the compiler inserts an empty no-argument constructor by default.

The compiler constructs an instance of the derived class in four steps in two distinct stages:

1. construct the base class portion of the complete object
    1. allocate memory for the instance variables in the order of their declaration
    2. execute the base class constructor
2. construct the derived class portion of the object

    3. allocate memory for the instance variables in the order of their declaration

4. execute the derived class constructor



In our example, let us define a no-argument constructor for the base class. The header file declares the no-argument constructor:

```
// Student.h

#include <iostream>
const int N = 30;
const int M = 13;

class Person {
    char person[N+1];
  public:
    Person();
    void set(const char* n);
    void display(std::ostream&) const;
};

class Student : public Person {
    int no;
    char grade[M+1];
  public:
    Student();
    Student(int, const char*);
    void display(std::ostream&) const;
};
```

The implementation file defines the base class constructor:

```
// Student.cpp

#include <cstring>
#include "Student.h"

Person::Person() {
    person[0] = '\0';
}

void Person::set(const char* n) {
```

```
    std::strncpy(person, n, N);
    person[N] = '\0';
}

void Person::display(std::ostream& os) const {
    os << person << ' ';
}

Student::Student() {
    no = 0;
    grade[0] = '\0';
}

Student::Student(int n, const char* g) {
    // insert validation logic here
    no = n;
    std::strcpy(grade, g);
}

void Student::display(std::ostream& os) const {
    Person::display(os);
    os << no << ' ' << grade;
}
```

The following client uses this implementation to produce the result shown on the right:

```
// Derived Class Constructors
// derivedCtors.cpp

#include <iostream>
#include "Student.h"

int main() {
    Person jane;
    Student harry(975, "ABBAD");

    harry.set("Harry");
    harry.display(std::cout);
    std::cout << std::endl;

    jane.set("Jane");
    jane.display(std::cout);
    std::cout << std::endl;
}
```

```
Harry 975 ABBAD

Jane
```

In this example, the compiler constructs the two objects as follows:

1. by allocating memory for jane
    1. by allocating memory for person
    2. the base class constructor initializes person to an empty string
2. by allocating memory for harry

1. by allocating memory for person
2. the base class constructor initializes person to an empty string
3. by allocating memory for no and grade
4. the derived class constructor initializes no and grade to 975 and "ABBAD" respectively

Passing Arguments to a Base Class Constructor

Each constructor of a derived class, other than the no-argument constructor, receives in its parameters all of the initial values passed by the client. Each constructor forwards any relevant values to the base class constructor. A base class constructor receives value in its parameters either from the derived class constructor or directly from the client program in the case of a base class object. The base class constructor uses the values received to complete building the base class part of the object. The derived class constructor uses the values received to complete building the derived class part of the object.

A call to the base class constructor from a derived class constructor that forwards values takes the form

```
Derived( parameters ) : Base( arguments )
```

where ***Derived*** is the name of the derived class and *Base* is the name of the base class. The single colon separates the header of the derived-class constructor from the call to the base class constructor. Omitting this call defaults to a call to the no-argument base class constructor.

Example

Let us replace the ***set()*** member function in the base class with a one-argument constructor and upgrade the ***Student's*** two-argument constructor to receive the student's name. The header file declares a single-argument base class constructor and a triple-argument derived class constructor:

```
// Student.h

 #include <iostream>
 const int N = 30;
 const int M = 13;

 class Person {
     char person[N+1];
   public:
```

```
    Person();
    Person(const char*);
    void display(std::ostream&) const;
};

class Student : public Person {
    int no;
    char grade[M+1];
  public:
    Student();
    Student(const char*, int, const char*);
    void display(std::ostream&) const;
};
```

The implementation of the single-argument constructor copies the name to the instance variable:

```
// Student.cpp

#include <cstring>
#include "Student.h"

Person::Person() {
    person[0] = '\0';
}

Person::Person(const char* nm) {
    std::strncpy(person, nm, N);
    person[N] = '\0';
}

void Person::display(std::ostream& os) const {
    os << person << ' ';
}

Student::Student() {
    no = 0;
    grade[0] = '\0';
}

Student::Student(const char* nm, int n, const char* g) : Person(nm) {
    // insert validation logic here
    no = n;
    std::strcpy(grade, g);
}

void Student::display(std::ostream& os) const {
    Person::display(os);
    os << no << ' ' << grade;
}
```

The following client uses this implementation to produce the output shown on the right:

```
// Derived Class Constructors with Arguments
 // drvdCtorsArgs.cpp
```

```
#include <iostream>
#include "Student.h"

int main() {
    Person jane("Jane");
    Student harry("Harry", 975, "ABBAD");

    harry.display(std::cout);
    std::cout << std::endl;

    jane.display(std::cout);
    std::cout << std::endl;
}
```

```
 Harry
975
ABBAD

 Jane
```

Inheriting Base Class Constructors

Consider cases where a derived class constructor forwards values to the base class constructor without executing any logic on the instance variables of the derived class. To avoid coding the almost empty derived class constructor, C++ lets us inherit the base class constructor directly.

The declaration for inheriting a base class constructor takes the form:

```
using Base::Base;
```

where ***Base*** is the name of the base class.

Example

Let us derive an Instructor class from the Person base class and inherit all of the constructors of the base class. The header file overrides the no-inheritance default:

```
// Student.h

 // compiles with GCC 4.8 or greater or equivalent

 #include <iostream>
 const int N = 30;
 const int M = 13;

 class Person {
     char person[N+1];
   public:
     Person();
     Person(const char*);
     void set(const char* n);
     void display(std::ostream&) const;
 };

 class Student : public Person {
     int no;
     char grade[M+1];
   public:
     Student();
     Student(const char*, int, const char*);
```

```
    void display(std::ostream&) const;
};

class Instructor : public Person {
  public:
    using Person::Person;
};
```

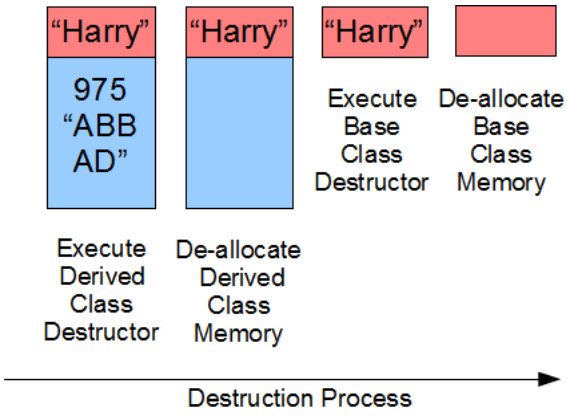
The implementation file remains unchanged. The following client uses this new class definition to produce the output shown on the right:

```
// Inherited Constructors
 // inheritCtors.cpp

#include <iostream>
#include "Student.h"

int main() {
    Instructor john("John");
    Person jane("Jane");
    Student harry("Harry", 975, "ABBAD");

    john.display(std::cout);
    std::cout << std::endl;

    harry.display(std::cout);
    std::cout << std::endl;

    jane.display(std::cout);
    std::cout << std::endl;
}
```

```
John

Harry 975 ABBAD

Jane
```

Destructors

A derived class does not inherit the destructor of its base class. Destructors execute in opposite order to the order of their object's construction. That is, the derived class destructor always executes before the base class destructor.



Example

Let us define destructors for our base and derived classes that insert messages to standard output. We declare the destructors in the class definitions:

```
// Student.h

 #include <iostream>
 const int N = 30;
 const int M = 13;

 class Person {
     char person[N+1];
   public:
     Person();
     Person(const char*);
     ~Person();
     void set(const char* n);
     void display(std::ostream&) const;
 };

 class Student : public Person {
     int no;
     char grade[M+1];
   public:
     Student();
     Student(const char*, int, const char*);
     ~Student();
     void display(std::ostream&) const;
 };
```

We specify the messages in the destructor definitions:

```
// Student.cpp

 #include <cstring>
 #include "Student.h"

 Person::Person() {
     person[0] = '\0';
 }

 Person::Person(const char* nm) {
     std::strncpy(person, nm, N);
     person[N] = '\0';
 }

 void Person::set(const char* n) {
     std::strncpy(person, n, N);
     person[N] = '\0';
 }

 Person::~Person() {
     std::cout << "Leaving " << person << std::endl;
 }
```

```
void Person::display(std::ostream& os) const {
    os << person << ' ';
}

Student::Student() {
    no = 0;
    grade[0] = '\0';
}

Student::Student(const char* nm, int n, const char* g) : Person(nm) {
    // insert validation logic here
    no = n;
    std::strcpy(grade, g);
}

Student::~Student() {
    std::cout << "\nLeaving " << no << std::endl;
}

void Student::display(std::ostream& os) const {
    Person::display(os);
    os << no << ' ' << grade;
}
```

The following client uses this implementation to produce the output shown on the right:

```
// Derived Class Destructors
 // drvdDtors.cpp

 #include <iostream>
 #include "Student.h"

 int main() {
     Person jane("Jane");
     Student harry("Harry", 975, "ABBAD");

     harry.display(std::cout);
     std::cout << std::endl;

     jane.display(std::cout);
     std::cout << std::endl;
 }
```

```
 Harry 975 ABBAD

 Jane
 Leaving 975
 Leaving Harry
 Leaving Jane
```

Helper Operators

A derived class does not support the helper functions of its base class. Each helper function is dedicated to the class that it supports. The compiler binds a call to a helper function on the basis of its parameter type(s).

Example

Let us upgrade our Student class to include overloads of the insertion and extraction operators for both base and derived classes. The header file contains:

```
// Student.h

#include <iostream>
const int N = 30;
const int M = 13;

class Person {
    char person[N+1];
  public:
    Person();
    Person(const char*);
    void display(std::ostream&) const;
};
std::istream& operator>>(std::istream&, Person&);
std::ostream& operator<<(std::ostream&, const Person&);

class Student : public Person {
    int no;
    char grade[M+1];
  public:
    Student();
    Student(const char*, int, const char*);
    void display(std::ostream&) const;
};
std::istream& operator>>(std::istream&, Student&);
std::ostream& operator<<(std::ostream&, const Student&);
```

The implementation file defines the helper operators:

```
// Student.cpp

#include <cstring>
#include "Student.h"

Person::Person() {
    person[0] = '\0';
}

Person::Person(const char* nm) {
    std::strncpy(person, nm, N);
    person[N] = '\0';
}

std::istream& operator>>(std::istream& is, Person& p) {
    char name[N+1];
    std::cout << "Name: ";
    is.getline(name, N+1);
    p = Person(name);
    return is;
}
```

```
std::ostream& operator<<(std::ostream& os, const Person& p) {
    p.display(os);
    return os;
}

void Person::display(std::ostream& os) const {
    os << person << ' ';
}

Student::Student() {
    no = 0;
    grade[0] = '\0';
}

Student::Student(const char* nm, int n, const char* g) : Person(nm) {
    // insert validation logic here
    no = n;
    std::strcpy(grade, g);
}

void Student::display(std::ostream& os) const {
    os << no << ' ' << grade;
}

std::istream& operator>>(std::istream& is, Student& s) {
    int no;
    char name[N + 1];
    char grade[M + 1];
    std::cout << "Name: ";
    is.getline(name, N+1);
    std::cout << "Number: ";
    is >> no;
    is.ignore(); // remove newline
    std::cout << "Grades: ";
    is.getline(grade, M+1);
    s = Student(name, no, grade);
    return is;
}

std::ostream& operator<<(std::ostream& os, const Student& s) {
    const Person& p = s; // copies base class address
    os << p;
    s.display(os);
    return os;
}
```

The following client uses this implementation to produce the output shown on the right:

```
// Helpers to Derived Classes
 // drvdHelpers.cpp

 #include <iostream>
 #include "Student.h"

 int main() {
     Person jane;
```

```
    Student harry;

    std::cin >> jane;
    std::cin >> harry;
    std::cout << jane << std::endl;
    std::cout << harry << std::endl;
}
```

```
Name: Jane Doe
Name: Harry
Number: 975
Grades: ABBD

Jane Doe
Harry 975 ABBD
```

1.5 Derived Classes and Resources

An inheritance hierarchy can access its resources at multiple levels. Managing relationships between the special member functions in a hierarchy with resources may require intervention. Our definitions of the copy constructors and copy assignment operators that manage resources may require explicit coding of the connections to their appropriate base class counterparts.

This chapter describes how to define the constructors and the copy assignment operator of derived classes that access resources and how to call their appropriate counterparts in the base classes.

Constructors and Destructor

Each constructor of a derived class with a resource calls a constructor of its base class. By default, that constructor is the no-argument constructor. To override this default, we insert an explicit call to the base class constructor.

The destructor of a derived class with a resource does not require any intervention to call the destructor of its base class, since each class in the hierarchy has but one destructor. The compiler inserts the call to the base class destructor.

Example

Let us upgrade the definition of our **Student** class to accommodate a client defined number of grades. We store the grades in dynamic memory and store the address of that memory in a resource instance pointer.

The upgraded definition of our **Student** class contains a resource instance pointer:

```
// Student.h

#include <iostream>
const int N = 30;

class Person {
    char person[N+1];
  public:
```

```
    Person();
    Person(const char*);
    ~Person();
    void display(std::ostream&) const;
};

class Student : public Person {
    int no;
    char* grade;
  public:
    Student();
    Student(const char*, int, const char*);
    ~Student();
    void display(std::ostream&) const;
};
```

Our two-argument constructor forwards the student's name to the single-argument constructor of the base class and then allocates memory for the grades. Our destructor deallocates that memory.

```
// Student.cpp

#include <cstring>
#include "Student.h"

Person::Person() {
    person[0] = '\0';
}

Person::Person(const char* nm) {
    std::strncpy(person, nm, N);
    person[N] = '\0';
}

Parson::Person() { }

void Person::display(std::ostream& os) const {
    os << person << ' ';
}

Student::Student() {
    no = 0;
    grade = nullptr;
}

Student::Student(const char* nm, int n, const char* g) : Person(nm) {
    // insert validation logic here
    no = n;
    if (g != nullptr) {
        grade = new char[std::strlen(g) + 1];
        std::strcpy(grade, g);
    }
    else
        grade = nullptr;
}
```

```
Student::~Student() {
    delete [] grade;
}

void Student::display(std::ostream& os) const {
    Person::display(os);
    os << no << ' ' << (grade != nullptr ? grade : "");
}
```

The conditional selection in Student::display() handles objects in a safe empty state.

The following client uses this implementation to produce the output shown on the right:

```
// Derived Class with a Resource
 // drvdResrce.cpp

#include <iostream>
#include "Student.h"

int main() {
    Person jane("Jane");
    Student harry("Harry", 975, "ABBAD");

    harry.display(std::cout);
    std::cout << std::endl;

    jane.display(std::cout);
    std::cout << std::endl;
}
```

```
Harry 975 ABBAD

Jane
```

Copy Constructor

The copy constructor of a derived class with a resource calls a constructor of the base class. By default, that constructor is the no-argument constructor. To override this default, we explicitly call the base class constructor of our choice.

The header in the definition of the copy constructor for a derived class takes the form

```
Derived(const Derived& identifier) : Base(identifier) {

    // ...
}
```

The parameter receives an unmodifiable reference to an object of the derived class. The argument in the call to the base class' constructor is the parameter's identifier.

Copying occurs in two distinct stages and four steps altogether:

1. copy the base class part of the existing object

1. allocate memory for the instance variables of the base class in the order of their declaration
2. execute the base class' copy constructor

2. copy the derived class part of the existing object
    3. allocate memory for the instance variables of the derived class in the order of their declaration
    4. execute the derived class' copy constructor

Example

Let us declare our own definition of the copy constructor for our **Student** class, but use the default definition for the **Person** class:

```
// Student.h

 #include <iostream>
 const int N = 30;

 class Person {
     char person[N+1];
   public:
     Person();
     Person(const char*);
     ~Person();
     void display(std::ostream&) const;
 };

 class Student : public Person {
     int no;
     char* grade;
   public:
     Student();
     Student(const char*, int, const char*);
     Student(const Student&);
     ~Student();
     void display(std::ostream&) const;
 };
```

We implement the copying steps as follows:

1. shallow copy the Person part of the source object
    - allocate static memory for person in the base class part of the newly created object
    - copy into person the string at address src.person
2. copy the Student part of the source object

- allocate static memory for no and *grade in the derived part of the newly created object
- shallow copy src.no into no
- allocate dynamic memory for a copy of src.grade
- deep copy the string at src.grade into grade

The default copy constructor for the base class performs a shallow copy. The copy constructor for the derived class calls the base class copy constructor and performs the deep copy itself:

```
// Student.cpp

 #include <cstring>
 #include "Student.h"

 Person::Person() {
     person[0] = '\0';
 }

 Person::Person(const char* nm) {
     std::strncpy(person, nm, N);
     person[N] = '\0';
 }

 Person::~Person() {}

 void Person::display(std::ostream& os) const {
     os << person << ' ';
 }

 Student::Student() {
     no = 0;
     grade = nullptr;
 }

 Student::Student(const char* nm, int n, const char* g) : Person(nm) {
     // insert validation logic here
     no = n;
     if (g != nullptr) {
         grade = new char[std::strlen(g) + 1];
         std::strcpy(grade, g);
     }
     else
         grade = nullptr;
 }

 Student::Student(const Student& src) : Person(src) {
     no = src.no;
     if (src.grade != nullptr) {
         grade = new char[std::strlen(src.grade) + 1];
         std::strcpy(grade, src.grade);
```

```
    }
    else
        grade = nullptr;
}

Student::~Student() {
    delete [] grade;
}

void Student::display(std::ostream& os) const {
    Person::display(os);
    os << no << ' ' << (grade != nullptr ? grade : "");
}
```

The Student copy constructor executes its logic after the Person copy constructor has executed its logic.

The following client uses this implementation to produce the output shown on the right:

```
// Derived Class with a Resource
 // drvdResrce.cpp

#include <iostream>
#include "Student.h"

int main() {
    Student harry("Harry", 975, "ABBAD"),
            harry_ = harry; // calls copy constructor

    harry.display(std::cout);
    std::cout << std::endl;

    harry_.display(std::cout);
    std::cout << std::endl;
}
```

```
Harry 975 ABBAD

Harry 975 ABBAD
```

Copy Assignment Operator

Any custom copy assignment operator of a derived class with a resource DOES NOT by default call the copy assignment operator of the base class. Only the default copy assignment operator of a derived class calls its base class counterpart. Accordingly in a custom copy assignment operator of a derived class with a resource, we need to call the base class copy assignmnet operator explicitly.

We call the base class copy assignment operator from within the body of the derived class assignment operator (unlike the copy constructor). The call can take one of the following forms:

- a functional expression
- an assignment to the base class part of the current object

The functional expression takes the form

```
Base::operator=(identifier);
```

The assignment expression takes the form

```
(Base&)*this = identifier;
```

Base is the name of the base class and ***identifier*** is the name of the right operand, which is the source object for the assignment. Note that the address of the derived object is the same as the address of the base class part of that object. The compiler distinguishes the call to the base class operator and a call to the derived class operator by the type of the left operand.

Example

The derived class definition declares a private member function for initializing along with the assignment operator:

```
// Student.h

#include <iostream>
const int N = 30;

class Person {
    char person[N+1];
  public:
    Person();
    Person(const char*);
    ~Person();
    void display(std::ostream&) const;
};

class Student : public Person {
    int no;
    char* grade;
    void init(int, const char*);
  public:
    Student();
    Student(const char*, int, const char*);
    Student(const Student&);
    ~Student();
    Student& operator=(const Student& src);
    void display(std::ostream&) const;
};
```

The private ***init()*** contains the copying logic shared by the constructors and the assignment operator:

```
// Student.cpp

#include <cstring>
#include "Student.h"
```

```
Person::Person() {
    person[0] = '\0';
}

Person::Person(const char* nm) {
    std::strncpy(person, nm, N);
    person[N] = '\0';
}

Person::~Person() {}

void Person::display(std::ostream& os) const {
    os << person << ' ';
}

Student::Student() {
    no = 0;
    grade = nullptr;
}

void Student::init(int n, const char* g) {
    no = n;
    if (g != nullptr) {
        grade = new char[std::strlen(g) + 1];
        std::strcpy(grade, g);
    }
    else
        grade = nullptr;
}

Student::Student(const char* nm, int n, const char* g) : Person(nm) {
    // insert validation logic here
    grade = nullptr;
    init(n, g);
}

Student::Student(const Student& src) : Person(src) {
    grade = nullptr;
    init(src.no, src.grade);
}

Student::~Student() {
    delete [] grade;
}

Student& Student::operator=(const Student& src) {
    if (this != &src) {
        // Base class assignment
        // 1 - functional expression
        // Person::operator=(src);
        // 2 - assignment expression
        (Person&)*this = src; // call base class assignment operator
        delete [] grade;
        init(src.no, src.grade);
    }
    return *this;
}
```

```
void Student::display(std::ostream& os) const {
    Person::display(os);
    os << no << ' ' << (grade != nullptr ? grade : "");
}
```

Sharing a **private** member function is one way of coding the copy constructor and assignment operator for the derived class.

The following client uses this implementation to produce the output shown on the right:

```
// Derived Class with a Resource Assignment
// drvdAssign.cpp

#include <iostream>
#include "Student.h"

int main() {
    Student harry("Harry", 975, "ABBAD"), backup;

    harry.display(std::cout);
    std::cout << std::endl;                                  Harry 975 ABBAD

    backup = harry;

    backup.display(std::cout);
    std::cout << std::endl;                                  Harry 975 ABBAD
}
```

Direct Call Copy Constructor

The alternative to sharing a member function is a direct call from the copy constructor to the assignment operator (as in the chapter entitled Classes and Resources). In a direct call design, we do not need to call the base class copy constructor since the assignment operator will copy the base class part of the object.

```
Student::Student(const Student& src) { // calls no-argument base class
constructor
    grade = nullptr;
    *this = src;
}

Student& Student::operator=(const Student& src) {
    if (this != &src) {
        // Base class assignment
        // 1 - functional expression
        // Person::operator=(src);
        // 2 - assignment expression
        Person& person = *this; // only copies address
        person = src;           // call base class operator
        delete [] grade;
        no = src.no;
        if (src.grade != nullptr) {
            grade = new char[std::strlen(src.grade) + 1];
            std::strcpy(grade, src.grade);
        }
```

```
        else
            grade = nullptr;
    }
    return *this;
}
```

1.6 Check Your Progress

Q.1 What are Derived Classes?

Q.2 What is the difference between Encapsulation and Inheritance?

Q.3 Define the following terms

i. Base Class

ii. Derived Class

Q.4 Giving an example How to pass Arguments to a Base Class Constructor?

Q.5 What is Copy Assignment Operator?

Block-5

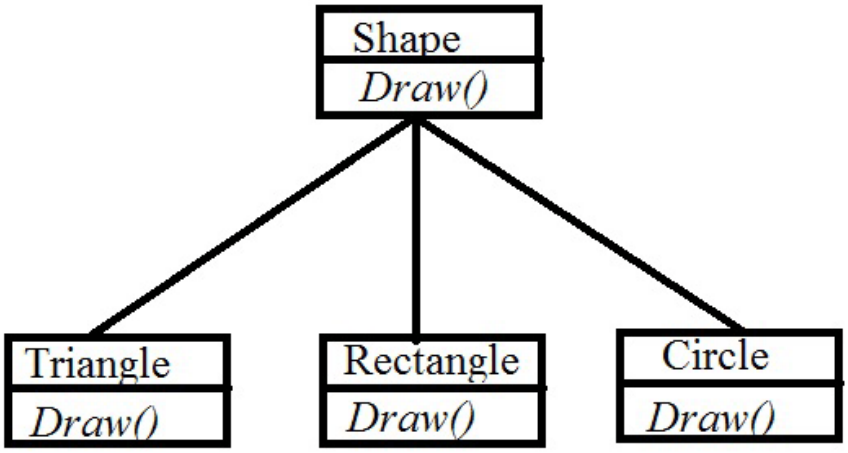
1.1 Learning Objectives

After going through this unit, the learner will able to:

- Understand the basic concepts of Polymorphism
- Learn about the virtual functions
- Define abstract base classes
- Learn about function templates

1.2 Introduction to Polymorphism

Polymorphism is another building block of object oriented programming. The philosophy that underlies is "one interface, multiple implementations." It allows one interface to control access to a general class of actions. Polymorphism can be achieved both in compile time and run time.



Polymorphism through virtual function

Virtual function : While declaring a function as virtual, the keyword virtual must precede the signature of the function. Every derived class redefines the virtual function for its own need.

Uses of virtual function enable run time polymorphism. We can use base class pointer to point any derived class object. When a base class contains a virtual function and base class pointer points to a derived class object as well as the derived class has a redefinition of base class virtual function, then the determination of which version of that function will be called is made on run

time. Different versions of the function are executed based on the different type of object that is pointed to.

The following example shows polymorphism through virtual function.

```
class Player {
	public:
		virtual void showInfo() {
			cout << "Player class info" << endl;
	}
};
class Footballer : public Player {
	public :
		void showInfo() {
			cout << "Footballer class info" << endl;
	}
};
class Cricketer : public Player {
	public :
		void showInfo() {
			cout << "Cricketer class info" << endl;
	}
};
int main() {
	Player *pPl,p1;
	pPl = &p1;
	Footballer f1;
	Cricketer c1;
	pPl->showInfo();
	pPl = &f1;
	pPl->showInfo();
	pPl = &c1;
	pPl->showInfo();
	system("pause");
	return 0;
}
```

In the above example, Player is the base class, Footballer and Cricketer are the derived class from Player. Virtual function showInfo is defined in Player class. Then it is redefined in Footballer and Cricketer class. Here, pPl is the Player class pointer, p1, f1 and c1 are Player, Footballer and Cricketer class object.

At first, pPl is assigned the address of p1, which is a base class object. If showInfo is now called using pPl, showInfo function of base class is executed. Next, pPl points to address derived class (Footballer & Cricketer) . If showInfo is called now, the redefined showInfo function of Footballer & Cricketer class are executed. The key point is, which version of the showInfo function will be executed depends on which object is currently pointed by base class pointer.

This decision is taken in run time, so it is an example of a run time polymorphism. This type of runtime polymorphism using virtual function is achieved by the base class pointer.

Function overloading

One way of achieving polymorphism is function overloading. When two or more functions share the same name with different parameter list, then this procedure is called function overloading and the functions are called overloaded function.

The following example shows polymorphism using function overloading.

```
class Player {
        string mName;
        int mAge;
        string mGameType;
        public:
                void setInfo(string str) {
                mName = str;
                cout << "Name :" << mName << endl;
        }
        void setInfo(string str, int age) {
                mAge = age;
                mName = str;
                cout << "Name :" << mName << " " << "Age :" << mAge << endl;
        }
        void setInfo(string str, int age, string game) {
                mAge = age;
                mName = str;
                mGameType = game;
                cout << "Name :" << mName  << " " << "Age :" << mAge << " " <<
"Game Type:" << mGameType << endl;
        }
};
int main() {
        Player p1;
        p1.setInfo("John Sena");
        p1.setInfo("C Ronaldo",25);
        p1.setInfo("J Kallis",38,"Cricket");
        return 0;
}
```

In this example, three functions have same name setInfo but different parameter list is defined for each. One function takes only one string parameter, another takes one string and one integer parameter and the last one takes two string and one integer as parameter. When we call setinfo function from Player class object p1, compiler looks at the argument list. It matches the argument list with the signature of three different funciton named setinfo, then one of the function is called according to the match. For example, when p1.setInfo("John Sena") is used, out of the three setinfo fucntion, the one with signature void setInfo(string str) is called and this

one is executed. When p1.setInfo("C Ronaldo",25)is used, out of the three setinfo function, the one with signature void setInfo(string str, int age)is called and this one is executed.

1.3 Overview of Polymorphism

Polymorphism is the third principal concept that object-oriented languages implement (alongside encapsulation and inheritance). Polymorphism refers to the multiplicity of meanings attached to an identifier. Polymorphic stands for 'of many forms'. A polymorphic language selects an operation from a multitude based on its object's type.

This chapter reviews the concept of *type* in detail along with its role in object-oriented languages. This chapter also describes the four categories of polymorphism that object-oriented languages support.

Languages

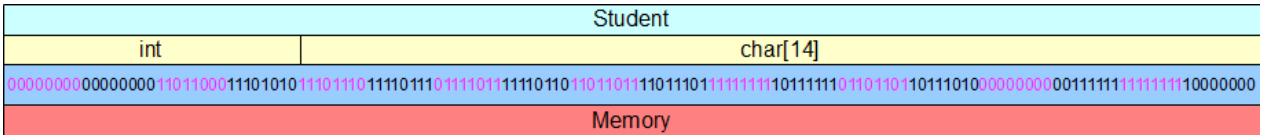
Programming languages evolved from untyped origins through monomorphic languages to polymorphic ones. Untyped languages support words of one fixed size. Assembly languages and BCPL are examples. Monomorphic languages support regions of memory of different sizes distinguished by their specific type. The type of an object, once declared, cannot change throughout the object's lifetime. Polymorphic languages relax this relation between a region of memory and the object's type by introducing ambiguity and bringing object descriptions closer to our natural language usage. The type of a polymorphic object may change during its lifetime.

Monomorphic languages require separate code for each type of object. For instance, a monomorphic language requires the programmer to code a separate ***sort()*** function for each data type, even though the logic is identical across all types. A polymorphic language, on the other hand, requires the programmer to code the function once and the langauge applies it to any type.

C++ assumes that an object is monomorphic by default, but lets the programmer explicitly identify it as polymorphic.

Types

Raw memory stores information in the form of bit strings. These bit strings may represent variables, objects, addresses, instructions, constants, etc. Without knowing what a bit string represents, the compiler does not know how to interpret its value. By associating a type with a region of memory, we inform the compiler how to interpret the bit string in that region of memory.



For example, if we associate a region of memory with a ***Student*** and define the structure of a ***Student***, the compiler knows that the first 4 bytes holds an ***int*** stored in equivalent binary form and the remaining 14 bytes hold an array of ***chars***.

Type System

A type system introduces consistency into a programming language. It is the first line of defense against coding relationships between unrelated entities. We assume that the entities in the expressions that we code do have some relation to one another. The presence of a type system enables the compiler to check whether the relation between entities follows a well-defined set of rules. Each type admits its own set of operations in forming expressions. The compiler rejects all operations outside this set. Breaking the type system exposes the underlying bits strings and introduces uncertainty in how to interpret the contents of a region of memory.

A strongly typed language enforces type consistency at compile-time and postpones type-checking to run-time only for polymorphic objects. C++ is a *strongly typed* langauge. It checks for type consistency on monomorphic objects at compile-time and on polymorphic objects at run-time.

Categories

Compilers apply their language's type system to identify possible violations of that system. Not all type differences between entities are necessarily errors. Those differences that the language allows expose its polymorphism. That is, the polymorphic features of a language represent the kinds of differences between types that the language supports.

Classifications

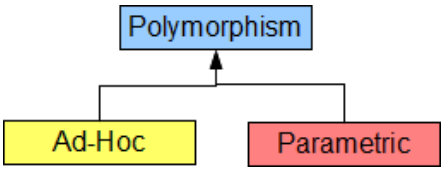
The polymorphic features of an object-oriented language are classified into four categories.

Christopher Strachey (1967) introduced the concept of polymorphism informally into procedural programming languages by distinguishing functions that work

- differently on different argument types
- uniformly on a range of argument types

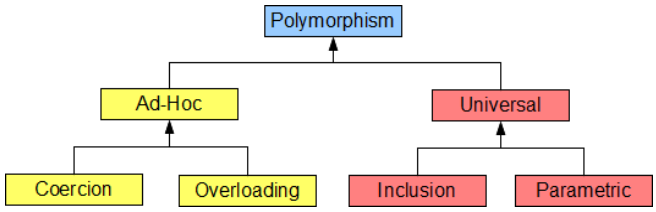
He defined the former as *ad-hoc* polymorphism and the latter as *parametric* polymorphism:

"Ad-Hoc polymorphism is obtained when a function works, or appears to work, on several different types (which may not exhibit a common structure) and may behave in unrelated ways for each type. Parametric polymorphism is obtained when a function works uniformly on a range of types; these types normally exhibit some common structure." (Strachey, 1967)



Cardelli and Wegner (1985) expanded Strachey's distinction to accommodate the object-oriented languages. They distinguished functions that work on

- a *finite* set of different and potentially unrelated types
    - coercion
    - overloading
- a potentially *infinite* number of types across some common structure
    - inclusion
    - parametric



Inclusion polymorphism is specific to object-oriented languages.

Ad-Hoc Polymorphism

Ad-hoc polymorphism is *apparent* polymorphism. Its polymorphic character disappears at closer scrutiny.

Coercion

Coercion covers convertible changes in argument type to match the type of a corresponding function parameter. It is a semantic operation to avoid a type error. If the compiler encounters a mismatch between the type of an argument and the type of the corresponding parameter, the language allows conversion from the type of the argument to the type of the corresponding parameter. The function definition itself only ever executes on one type - that of its parameter.

C++ implements coercion at compile time. If a compiler succeeds in matching the type of an argument to the type of the corresponding parameter in the function call, the compiler inserts the conversion code immediately before the function call.

Coercion may

- narrow the argument type (narrowing)
- widen the argument type (promotion)

For example,

```
// Ad-Hoc Polymorphism - Coercion
 // polyCoercion.cpp

 #include <iostream>

 // One function definition:

 void display(int a) const {
     std::cout << "One argument (" << a << ')';
 }

 int main( ) {

     display(10);
     std::cout << std::endl;
     display(12.6); // narrowing
     std::cout << std::endl;
     display('A'); // promotion
     std::cout << std::endl;
 }
```

```
One argument (10)

One argument (12)

One argument (63)
```

Most programming languages support coercion to some extent. For instance, C narrows and promotes argument types in function calls so that the same function will accept a variety of argument types, albeit limited.

Overloading

Overloading covers accepted variations in a function's definition to match the argument types to corresponding parameter types. It is a syntactic abbreviation that associates the same function identifier with a variety of function definitions by distinguishing its parameter sets. The same function name can be used with a variety of unrelated argument types. Each set of argument types has its own function definition. The compiler binds the function call to the matching function definition.

Unlike coercion, overloading does not involve any common logic shared by the function definitions with the same identifer. Uniformity is a coincidence rather than the rule. The definitions may contain totally unrelated logic. Each definition works only on its set of types. The number of overloaded functions is limited by the number of definitions implemented in the source code.

C++ compilers implement overloading at compile time by converting the function definitions into functions with different identifiers: the language mangles the original identifier with the paramter types to generate an unique name. The linker uses the mangled name to bind the function call to the appropriate function definition.

For example,

```
// Ad-Hoc Polymorphism - Overloading
 // polymorphismOverloading.cpp

 #include <iostream>

 // Two function definitions:

 void display() const {
     std::cout << "No arguments";
 }

 void display(int a) const {
     std::cout << "One argument (" << a << ')';
 }

 int main( ) {

     display();
     std::cout << std::endl;
     display(10);
     std::cout << std::endl;
 }
```

```
 No arguments

 One argument (10)
```

The C language does not admit overloading and requires a unique name for each function definition.

Universal Polymorphism

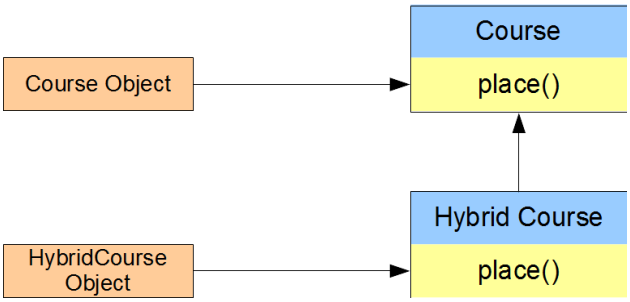
Universal polymorphism is *true* polymorphism. Its polymorphic character survives at closer scrutiny.

Universal polymorphism imposes no restriction on the admissible types. The same function (logic) applies to a potentially unlimited range of different types.

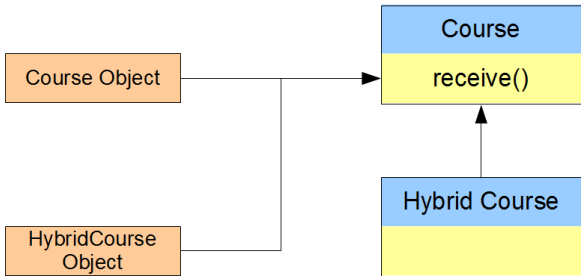
Inclusion

Inclusion polymorphism covers selection of a member function definition from a set of definitions based on an object's type. The type is one of the types belonging to an inheritance hierarchy. The term *inclusion* refers to one type including another type within the hierarchy. All function definitions share the same name throughout the hierarchy.

In the hierarchy illustrated below, a ***HybridCourse*** uses one mode of delivery while a ***Course*** uses another mode. That is, ***a mode()*** query on a ***Course*** object reports a different result from a ***mode()*** query on a ***HybridCourse*** object.



Operations that are identical for all of the types within the hierarchy require only single definitions. The ***assessments()*** query on a ***HybridCOurse*** object invokes the same logic as one on the ***Course*** object. Defining a query for the ***HybridCourse*** class would only duplicate existing code. Inclusion polymorphism eliminates duplicate logic across a hierarchy.



Parametric

Parametric polymorphism covers definitions that share identical logic independently of type. The logic is common to all possible types, without restriction. The types need not be related in any way. For example, a function that sorts ints uses the same logic as a function that sorts doubles. If we have already written a function to sort ints, what would we do to create a function that sorts doubles. C++ implements parametric polymorphism at compile-time using template syntax.

1.4 Virtual Functions

An object-oriented language implements inclusion polymorphism through sets of member functions on polymorphic objects. Polymorphic objects can change their type throughout their lifetime. To track these changes we distinguish between static and dynamic types. An object's static type is its reference type and a compile-time constant. The object's dynamic type is its current type and only known at run-time.

C++ implements polymorphic objects using pointers. The pointer type identifies the static type and the inheritance hierarchy that contains the set of member functions. The dynamic type is the type that we use to allocate memory for the object.

This chapter describes the way in which C++ implements inclusion polymorphism, which includes the binding of a function call to a member function definition depending on an object's static or dynamic type.

Function Bindings

The compiler uses an object's type to bind a function call to a function definition. The object's type determines which member function in the inheritance hierarchy to call.

Function binding takes either of two forms:

- early binding - based on the object's static type
- dynamic dispatch - based on the object's dynamic type

Early Binding

Consider the following definition of our ***Student*** class.

```
// Student.h

 #include <iostream>
 const int N = 30;
 const int M = 13;

 class Person {
     char person[N+1];
   public:
     Person();
     Person(const char*);
     void display(std::ostream&) const;
 };
```

```
class Student : public Person {
    int no;
    char grade[M+1];
  public:
    Student();
    Student(const char*, int, const char*);
    void display(std::ostream&) const;
};
```

The implementation file is also the same as in the chapter entitled Functions in a Hierarchy:

```
// Student.cpp

#include <cstring>
#include "Student.h"

Person::Person() {
    person[0] = '\0';
}

Person::Person(const char* nm) {
    std::strncpy(person, nm, N);
    person[N] = '\0';
}

void Person::display(std::ostream& os) const {
    os << person << ' ';
}

Student::Student() {
    no = 0;
    grade[0] = '\0';
}

Student::Student(const char* nm, int n, const char* g) : Person(nm) {
    // insert validation logic here
    no = n;
    std::strcpy(grade, g);
}

void Student::display(std::ostream& os) const {
    Person::display(os);
    os << no << ' ' << grade;
}
```

The **main()** function in the client code listed below calls the global ***show()*** function twice, first for a ***Student*** object and second for a ***Person*** object. The compiler binds both calls to the ***Person*** version of ***display()*** irrespective of argument type in the call itself. That is, the compiler uses the parameter type in ***show()*** to determine which member function definition to call. We call this an *early binding*. The client program produces the output shown on the right:

```
// Early Binding
 // earlyBinding.cpp

 #include <iostream>
 #include "Student.h"

 void show(const Person& person) {
     person.display(std::cout);
     std::cout << std::endl;
 }

 int main() {
     Student harry("Harry", 975, "ABBAD");
     Person jane("Jane Doe");

     show(harry);
     show(jane);
 }
```

```
 Harry
 Jane Doe
```

This is the most efficient binding of a call to a member function's definition, since it occurs at compile-time. Early binding is the default in C++.

Note that no shadowing occurs inside the global ***show()*** function**.** ***show()*** has no way of knowing which version of ***display()*** to select beyond the type of its parameter. (To demonstrate shadowing, add the statements ***harry.display()*** and ***jane.display()*** to the ***main()*** function.)

Dynamic Dispatch

To call the member function associated with an object's dynamic type, we must wait until run-time, when the executable code is aware of the object's dynamic type. We call this dynamic dispatching.

C++ provides the keyword ***virtual*** for overriding the default early binding. If this keyword is present, the compiler inserts code that binds the call to most derived version of the member function based on the object's dynamic type.

For example, the keyword ***virtual*** in the following class definition instructs the compiler to postpone calling the ***display()*** member function definitions until run-time:

```
// Student.h

 #include <iostream>
 const int N = 30;
 const int M = 13;

 class Person {
     char person[N+1];
   public:
     Person();
     Person(const char*);
```

```
    virtual void display(std::ostream&) const;
};

class Student : public Person {
    int no;
    char grade[M+1];
  public:
    Student();
    Student(const char*, int, const char*);
    void display(std::ostream&) const;
};
```

Although the implementation file and the client program have not changed, the ***show()*** function now calls the most derived version of ***display()*** based on the type of the argument passed to it and produces the output shown on the right:

```
// Late Binding
 // lateBinding.cpp

 #include <iostream>
 #include "Student.h"

 void show(const Person& person) {
     person.display(std::cout);
     std::cout << std::endl;
 }

 int main() {
     Student harry("Harry", 975, "ABBAD");
     Person jane("Jane Doe");

     show(harry);
     show(jane);
 }
```

```
Harry 975 ABBAD
Jane Doe
```

show(harry) passes an unmodifiable reference to a ***Student***, while ***show(jane)*** passes an unmodifiable reference to a ***Person***. In each case, the version of ***display()*** bound at run time is the most derived version for the type referenced by the parameter in ***show().***

Note that if we passed the argument to the ***show()*** function by value instead of reference, the ***show()*** function would still call the most derived version of ***display(),*** except that that version would be the Person version in both cases.

Overriding Dynamic Dispatching

To override dynamic dispatching with an early binding, we specify the scope explicitly:

```
void show(const Person& person) {
     person.Person::display(std::cout);
     std::cout << std::endl;
 }
```

Documentation

We can identify a member function as **virtual** even if no derived class exists. Some programmers include the qualifier **virtual** in derived class prototypes as a form of documentation. This improves readability but has no syntactic effect.

Polymorphic Objects

A polymorphic object's static type identifies the hierarchy to which the object belongs, regardless of its current dynamic type. We specify the object's static type through

- a pointer declaration
- a receive-by-address parameter
- a receive-by-reference parameter

For example, the highlighted code specifies the static type pointed to by ***person***:

```
// Polymorphic Objects - Static Type

 #include <iostream>
 #include "Student.h"

 int main() {
     Person* person = nullptr;

     // ...

 }
```

We specify the object's dynamic type on the constructor that allocates memory for it. The region of memory that the object occupies depends on that type.

The highlighted code specifies the dynamic type. The results produced by this code are listed on the right:

```
// Polymorphic Objects - Dynamic Type
 // dyanmicType.cpp

 #include <iostream>
 #include "Student.h"

 void show(const Person& person) {
     person.display(std::cout);
     std::cout << std::endl;
 }

 int main() {
     Person* person = nullptr;

     person = new Person("Jane Doe");
     show(*person);                                    Jane Doe
```

```
    delete person;

    person = new Student("Harry", 975, "ABBAD");
    show(*person);                                    Harry 975 ABBAD
    delete person;
}
```

In ***main(), person*** initially points to a ***Person*** (the static type) with no memory allocated. At this point the object's dynamic type is unknown. After the first allocation, ***person*** points to a ***Student*** type (dynamic type). After the second allocation, ***person*** points to a ***Person*** type (the new dynamic type). The static and dynamic types are related through the hierarchy.

We only need one ***show()*** function to display both dynamic types.

person points to a polymorphic object throughout its lifetime. The object may assume any an infinite number of types as long as each is derived from a ***Person*** type. ***show()*** is a polymorphic function with a parameter that receives an unmodifiable reference to an infinite number of types derived from a ***Person*** type.

Virtual Destructors

If a derived class acquires a resource, its own destructor typically releases that resource. To ensure that the object calls the derived class' destructor at the end of its scope, we declare the base class destructor virtual. Since a derived class destructor automatically calls its immediate base class' destructor, all destructors in the object's hierarchy will be called in turn.

Good Design

Good design codes the destructor in a base class as ***virtual***, even if no class is currently derived from that base class. The presence of a ***virtual*** base class destructor ensures that the most derived destructor will be called if and when a class is derived from the base class without requiring an upgrade to the definition of the base class.

Efficiency and Flexibility

Implementing inclusion polymorphism produces efficient and flexible code. That is, virtual functions reduce code size considerably. Our ***show()*** function works on objects of any type within the ***Person*** hierarchy. We only define member functions (***display()***) for those classes requiring distinct processing.

During the lifecycle of a client application that uses our hierarchy, we may add several classes to the hierarchy. Our original coding, without any alteration, selects the most derived version of the member function in each upgrade of the hierarchy.

1.5 Abstract Base Classes

Separating interfaces from implementations promotes low coupling in object-oriented designs. An interface specifies what an object offers to a client, while an implementation specifies how it does what it offers. This distinction is quite useful in designing class hierarchies. An interface to a class hierarchy identifies what the hierarchy offers, while the implementation describes how each class delivers what the hierarchy offers. The interface effectively hides the hierarchy from any client. We can upgrade the hierarchy by adding derived classes without having to change the client code.

C++ supports this distinction through abstract base classes and concrete classes. An abstract base class is a class without an implementation. It defines an interface. A concrete class is a derived class that implements the interface. An abstract base class identifies the member functions that the class hierarchy exposes to its client but not their definitions.

This chapter describes the principal components of an abstract base class and shows how to define an abstract base class in terms of them. Abstract base classes are gateways to testing their inheritance hierarchies. This chapter includes an example of a unit test on an abstract base class.

Pure Virtual Functions

The principal component of an abstract base class is a *pure virtual member function*. *Pure* refers to the lack of implementation details. We specify its signature but omit its definition. A client only requires this signature to access whatever implementation matches its object's dynamic type.

Declaration

The declaration of a pure virtual function takes the form

```
virtual Type identifier(parameters) = 0;
```

The assignment to 0 identifies the function as pure. A pure function must be a virtual member function.

For example, let us specify that the **Person** hierarchy provides some implementation of a display function with the signature **display() const**. The pure virtual function that exposes this functionality is:

```
class iPerson {
   public:
     virtual void display(std::ostream&) const = 0;
 };
```

Set of Implementations

The definitions of the member functions throughout the class hierarchy with the same signature as a pure virtual function provide the different implementations available to the client.

Abstract Base Classes

An abstract base class is a class that contains or inherits a pure virtual function that has yet to be defined. Any attempt to create an instance of an abstract base class generates a compiler error.

Definition

An abstract base class is a set of pure virtual member functions. The class definition contains their declarations. We call an abstract base class without data members an *interface*.

Example

Let us define an abstract base class named **iPerson** for our **Person** hierarchy and expose the **display()** member function to any client that accesses the interface.

The **iPerson.h** header file contains the definition of our abstract class:

```
// Abstract Base Class for the Person Hierarchy
 // iPerson.h

 #include <iostream>

 class iPerson {
   public:
     virtual void display(std::ostream&) const = 0;
 };
```

We derive our **Person** class from this interface. The header file for our **Person** and **Student** class definitions includes the header file that defines our abstract base class:

```
// Student.h

 #include "iPerson.h"
 const int N = 30;
 const int M = 13;

 class Person : public iPerson {
     char person[N+1];
```

```
  public:
    Person();
    Person(const char*);
    void display(std::ostream&) const;
};

class Student : public Person {
    int no;
    char grade[M+1];
  public:
    Student();
    Student(const char*, int, const char*);
    void display(std::ostream&) const;
};
```

Declaring **display()** a member function of both **Person** and **Student** informs the compiler that each concrete class will implement a version of this member function or access a previously implemented version.

Unit Tests on an Interface

It is good programming practice to code unit tests for an interface rather than any implementation. This approach assumes that the interface does not change. We can then perform unit tests on the interface at every upgrade throughout an object's lifecycle.

Sorter Classes

Consider an interface that exposes the sort() member function of a hierarchy of Sorter classes. The Sorter module contains all of the implemented algorithms. The interface and the tester module remain unchanged. With every upgrade to the Sorter module, we can rerun the test suite on the interface.

The header file for our Sorter module contains:

```
// iSorter.h

class iSorter {
  public:
    virtual void sort(float*, int) = 0;
};

class SelectionSorter : public iSorter {
  public:
    void sort(float*, int);
};

class BubbleSorter : public iSorter {
  public:
    void sort(float*, int);
};
```

The header file for the ***Tester*** module contains:

```
// Tester.h

 class iSorter;

 void test(iSorter*, float*, int, const char*);
```

The implementation file for the **Tester** module contains:

```
// Tester for Sorter Interface
 // tester.cpp

 #include <iostream>
 #include "iSorter.h"

 void test(iSorter* sorter, float* a, int n, const char* msg) {
     sorter->sort(a, n);
     bool sorted = true;
     for (int i = 0; i < n - 1; i++)
         if (a[i] > a[i+1]) sorted = false;
     if (sorted)
         std::cout << msg << " is sorted" << std::endl;
     else
         std::cout << msg << " is not sorted" << std::endl;
 }
```

The implementation file for the **Sorter** module defines **the sort()** member functions for the **SelectionSorter** class and the **BubbleSorter** class:

```
// Sorter.cpp

 #include "iSorter.h"

 void SelectionSorter::sort(float* a, int n) {
     int i, j, max;
     float temp;

     for (i = 0; i < n - 1; i++) {
         max = i;
         for (j = i + 1; j < n; j++)
             if (a[max] > a[j])
                 max = j;
         temp = a[max];
         a[max] = a[i];
         a[i] = temp;
     }
 }

 void BubbleSorter::sort(float* a, int n) {
     int i, j;
     float temp;

     for (i = n - 1; i > 0; i--) {
         for (j = 0; j < i; j++) {
             if (a[j] > a[j+1]) {
                 temp = a[j];
```

```
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
}
```

The following program uses this **Sorter** implementation to produce the output shown on the right:

```
// Test Main for Sorter Interface
 // test_main.cpp

 #include <iostream>
 #include <ctime>
 #include "Tester.h"
 #include "iSorter.h"

 void populate(float* a, int n) {
     srand(time(nullptr));
     float f = 1.0f / RAND_MAX;
     for (int i = 0; i < n; i++)
         a[i] = rand() * f;
 }

 int main() {
     int n;
     std::cout << "Enter no of elements : ";
     std::cin >> n;
     float* array = new float[n];

     iSorter* sorter = nullptr;

     sorter = new SelectionSorter();
     populate(array, n);
     test(sorter, array, n,
"SelectionSort");
     delete sorter;

     sorter = new BubbleSorter();
     populate(array, n);
     test(sorter, array, n, "BubbleSort");
     delete sorter;

     delete [] array;
 }
```

```
Enter no of elements : 1000
```

```
 SelectionSort is sorted
```

```
 BubbleSort is sorted
```

1.6 Function Templates

Parametric polymorphism perfects the separation of interfaces from implementations. The object's type and the logic executed on that type are completely independent. Different clients can access the same logic using different totally unrelated types.

C++ implements parametric polymorphism using template syntax. The compiler generates the implementation for each client type at compile-time from the template that we define.

This chapter describes how to implement parametric polymorphism using templax syntax with particular reference to functions and how to define an explicit specialization of a template for a particular type. This chapter also describes the templated keywords available for casting of values from one type to another and compares these constrained casts to the older unconstrained casts.

Template Syntax

The definition of a template resembles that of a global function with the parentheses replaced by angle brackets. A template header takes the form

```
template<Type identifier[, ...]>
```

The keyword **template** identifies the subsequent code block as a template. The less-than greater-than angle bracket pair (< >) encloses the template's parameter definitions. The ellipsis stands for more comma-separated parameters.

Each parameter declaration consists of a type and an identifier. *Type* may be any of

- typename - to identify a type (fundamental or compound)
- class - to identify a type (fundamental or compound)
- int, long, short, char - to identify a non-floating-point fundamental type
- a template parameter

identifier is a placeholder for the argument specified by the client.

For example,

```
template <typename T>
 // ... template body follows here
     T value; // value is of type T
```

```
 template <class T>
 // ... template body follows here
     T value; // value is of type T
```

The compiler replaces **T** with the argument specified by the client.

Complete Definition

Consider a function that swaps values in two different memory locations. The code for two int variables may be defined using references:

```
void swap(int& a, int& b) {
     int c;
     c = a;
     a = b;
     b = c;
 }
```

The template for all functions that swap values in this way follows from replacing the specific type int with the type variable T and inserting the template header:

```
// Template for swap
 // swap.h

 template<typename T>
 void swap(T& a, T& b) {
     T c;
     c = a;
     a = b;
     b = c;
 }
```

We define templates in header files; in this case, in **swap.h.**

Call of the Implementation

The call to a templated function determines the specialization that the compiler will generate. The compiler binds the call to that specialization.

For example, to call the **swap()** function for two **doubles** and two l**ongs**, we write the following and leave the remaining work to the compiler:

```
// Calling a Templated Function
 // swap.cpp

 #include <iostream>
 #include "swap.h" // template
definition

 int main() {
     double a = 2.3;
     double b = 4.5;
     long   d = 78;
     long   e = 567;

     swap(a, b); // compiler
generates
                 // swap(double,      Swapped values are 4.5 and 2.3
```

```
double)

     std::cout << "Swapped values
are " <<
          a << " and " << b <<
std::endl;                                Swapped values are 567 and 78

     swap(d, e); // compiler
generates
                 // swap(long, long)

     std::cout << "Swapped values
are " <<
          d << " and " << e <<
std::endl;
 }
```

The arguments in each call are unambiguous in their type and the compiler can specialize the template appropriately. If the arguments are ambiguous, the compiler reports an error.

Explicit Specialization

A template definition may have exceptions for certain arguments. We define separate specializations for each exception not covered by the template definition.

Example

The following template definition return the maximum of two arguments and applies to any fundamental type:

```
// Maximum Function
 // maximum.h

 template<typename T>
 T maximum(T a, T b) {
     return a > b ? a : b;
 }
```

To accomodate the const char* type, we specialize the template explicitly. An explicit specialization has an empty paramter list:

```
// Maximum Function
 // + explicit specialization for const char*
 // maximum.h

 template<typename T>
 T maximum(T a, T b) {
     return a > b ? a : b;
 }

 template<> // explicit specialization - empty parameter list
 const char* maximum<const char*>(const char* a, const char* b) {
     char* largest = a;
```

```
    bool done = false;
    for (int i = 0; !done && a[i] != '\0' && b[i] != '\0'; i++) {
        done = a[i] != b[i];
        if (b[i] > a[i]) largest = b;
    }
    return largest;
}
```

We place explicit specializations after their general template definition.

The compiler binds the second call to **maximum** to the explicit specialization for the **const char*** type:

```
// Maximum of Two Strings
 // maximum.cpp

 #include <iostream>
 #include "maximum.h"

 int main() {
    double a = 2.3;
    double b = 4.5;
    const char d[4] = "abc";
    const char e[4] = "def";

    double c = maximum(a, b);

    std::cout << "Greater of " <<    Greater of 2.3, 4.5 is 4.5
         a << ", " << b <<
         " is " << c << std::endl;

    const char* f = maximum(d, e);

    std::cout << "Greater of " <<    Greater of abc, def is def
         d << ", " << e <<
         " is " << f << std::endl;
 }
```

Class Template

The syntax for class templates is similar to that for function templates.

The following template defines **Array** classes of specified size in static memory. The template parameters are the element type **(T)** and the size of the array **(N):**

```
// Template for Array Classes
 // Array.h

 template <class T, int N>
 class Array {
    T a[N];
 public:
    T& operator[](int i) { return a[i]; }
 };
```

The compiler generates a class definition for element type int and array size 5 using the Array template definition. The output from executing this client program is shown on the right:

```
// Class Template
 // Template.cpp

 #include <iostream>
 #include "Array.h"

 int main() {
     Array<int, 5> a, b;

     for (int i = 0; i < 5; i++)
         a[i] = i * i;

     b = a;

     for (int i = 0; i < 5; i++)
         std::cout << b[i] << ' ';
     std::cout << std::endl;
}
```

```
 0 1 4 9 16
```

Type Casting

Type safety is an important feature of any strongly typed language. Bypassing the type system introduces ambiguity to the language itself and is best avoided. Type casting a value from one type to another type circumvents the type system's type checking facilities. We implement casts only where absolutely unavoidable and localize them as much as possible.

C++ supportsconstrained type casting through template syntax using one of the following keywords:

- **static_cast<*Type*>(*expression*)**
- **reinterpret_cast<*Type*>(*expression*)**
- **const_cast<*Type*>(*expression*)**
- **dynamic_cast<*Type*>(*expression*)**

Type refers to the destination type. ***expression*** refers to the value being cast to the destination type.

Related Types

The **static_cast<*Type*>(*expression*)** keyword converts the expression from its evaluated type to the specified type. By far, this is the most common form of constrained cast.

For example, to cast **minutes** to a **float** type, we write:

```
// Cast to a Related Type
 // static_cast.cpp
```

```
 #include <iostream>

 int main() {
     double hours;
     int minutes;

     std::cout << "Enter minutes : ";
     std::cin >> minutes;
     hours = static_cast<double>(minutes)/ 60;  // int and float are related
     std::cout << "In hours, this is " << hours;
 }
```

static_cast<*Type*>(*expression*) performs limited type checking. It rejects conversions between pointer and non-pointer types.

For example, the following cast generates a compile-time error:

```
#include <iostream>

 int main() {
     int x = 2;
     int* p;

     p = static_cast<int*>(x);  // FAILS: unrelated types

     std::cout << p;
 }
```

Some static casts are portable across different platforms.

Unrelated Types

The **reinterpret_cast<*Type*>(*expression*)** keyword converts the expression from its evaluated type to an unrelated type. This cast may produce a value that has the same bit pattern as the evaluated expression.

For example, to cast an **int** type to a pointer to an **int** type, we write:

```
// Cast to an Unrelated Type
 // reinterpret_cast.cpp

 #include <iostream>

 int main( ) {
     int x = 2;
     int* p;

     p = reinterpret_cast<int*>(x);  // int and int* are unrelated

     std::cout << p;
 }
```

reinterpret_cast<*Type*>(*expression*) performs minimal type checking. It rejects conversions between related types.

For example, the following cast generates a compile-time error:

```
#include <iostream>

 int main( ) {
     int x = 2;
     double y;

     y = reinterpret_cast<double>(x);  // FAILS types are related

     std::cout << y;
 }
```

Few reinterpret casts are portable. Uses include

- evaluating raw data
- recovering data where types are unknown
- quick and messy calculations

Unmodifiable Types

The **const_cast<*Type*>(*expression*)** keyword removes the **const** status from an expression.

A common use case is a function written by another programmer that does not receive a const parameter but should receive one. If we cannot call the function with a const argument, we temporarily remove the const status and hope that the function is truly read only.

```
// Strip const status from an Expression
 // const_cast.cpp

 #include <iostream>

 void foo(int* p) {
     std::cout << *p << std::endl;
 }

 int main( ) {
     const int x = 3;
     const int* a = &x;
     int* b;

     // foo expects int* and not const int*
     b = const_cast<int*>(a);  // remove const status
     foo(b);
 }
```

const_cast<*Type*>(*expression*) performs minimal type checking. It rejects conversions between different types.

For example, the following code generates a compile-time error:

```
#include <iostream>

 int main( ) {
     const int x = 2;
     double y;

     y = const_cast<double>(x); // FAILS

     std::cout << y;
 }
```

Inherited Types

The **dynamic_cast<Type>(expression)** keyword converts the value of an expression from its current type to another type within the same class hierarchy.

For example, to cast a pointer to derived object d to a pointer to its base class part, we write:

```
// Cast to a Type within
the Hierarchy
 // dynamic_cast.cpp

 #include <iostream>

 class Base {
   public:
     void display() const
{ std::cout << "Base\n"; }
 };
 class Derived : public
Base {
   public:
     void display() const
{ std::cout <<
"Derived\n"; }
 };

 int main( ) {
     Base* b;
     Derived* d = new
Derived;
                            Base
     b =
dynamic_cast<Base*>(d);    Derived
// in the same hierarchy
     b->display();
     d->display();
     delete d;
 }
```

dynamic_cast<Type>(expression) performs some type checking. It rejects conversions from a base class pointer to a derived class pointer if the object is monomorphic.

For example, the following cast generates a compile-time error:

```
#include <iostream>

 class Base {
   public:
     void display() const { std::cout << "Base\n"; }
 };
 class Derived : public Base {
   public:
     void display() const { std::cout << "Derived\n"; }
 };

 int main( ) {
     Base* b = new Base;
     Derived* d;

     d = dynamic_cast<Derived*>(b);  // FAILS
     b->display();
     d->display();
     delete d;
 }
```

Note that a **static_cast** works here and may produce the result shown on the right. However, the **Derived** part of the object would then be incomplete**. static_cast** does not check if the object is complete, leaving the responsibility to the programmer.

```
#include <iostream>

 class Base {
   public:
     void display() const
{ std::cout << "Base\n"; }
 };
 class Derived : public
Base {
   public:
     void display() const
{ std::cout <<
"Derived\n"; }
 };

 int main( ) {
     Base* b = new Base;
     Derived* d;
                              Base
     d =
static_cast<Derived*>(b);  Derived
// OK
     b->display();
     d->display();
```

```
    delete d;
}
```

Old-Style Casts

C++ inherited its original casting facilities from C and built directly on them. The constrained syntax described above is more discriminating than the older syntax, which remains in the language for legacy reasons. The availability of these older features allows programmers to bypass the type system and directly weaken the compiler's ability to identify type errors.

For example, even though converting from an int to a pointer to an int is most probably a typing mistake, C and hence C++ allow:

```
    int x = 2;
    int* p;
    p = (int*)(x); // MOST PROBABLY A TYPING ERROR (& missing)!
```

Such syntax is nearly always an error that we expect the type-checking system to trap. Errors that result from such casts are very difficult to find when embedded within thousands of lines of code.

C++ supports old-style casting in two distinct forms - plain C-style casts and C++-function-style casts:

```
(Type) identifier       and      Type (identfier)
```

These forms are interchangeable for fundamental types, but not pointer types. For conversions to pointer types, only the C-style cast is available.

C-Style Casts

To cast a value from one type to another using a C-style cast, we preface the identifier with the name of the target type enclosed in parentheses:

```
// C-Style Casting
 // c_cast.cpp

 #include <iostream>

 int main() {
    double hours;
    int minutes;
    std::cout << "Enter minutes : ";
    std::cin >> minutes;
    hours = (double) minutes / 60;  // C-Style Cast
    std::cout << "In hours, this is " << hours;
 }
```

Function-Style Casts

To cast a value from one type to another using a function-style cast, we enclose in parentheses the variable or object whose value we wish to cast to the target type:

```
// Function Style Casting
 // functionStyleCast.cpp

 #include <iostream>

 int main() {
     double hours;
     int minutes;
     std::cout << "Enter minutes : ";
     std::cin >> minutes;
     hours = double(minutes) / 60;  // Function-Style Cast
     std::cout << "In hours, this is " << hours;
 }
```

Comparison

The old-style casts (for example, **(int)x)** apply without regard to the category of the conversion. This syntax does not convey the programmer's intent. An old-style cast can mean any of the following:

- **static_cast**
- **const_cast**
- **static_cast + const_cast**
- **reinterpret_cast**
- **reinterpret_cast + const_cast**

The constrained casts on the other hand by distinguishing the different categories improve the degree of type checking.

It is always safer type-wise to code a **static_cast** rather than a C-style cast.

1.7 Check Your Progress

Q.1 What is function overloading? Explain with the help of an example.
Q.2 What is Polymorphism? Explain the types of polymorphism with an example.
Q.3 Define the following terms
    i. Ad-Hoc Polymorphism
    ii. Universal Polymorphism
Q.4 What is virtual function? Explain with the help of an example.