CHAPTER 15

POLYMORPHISM, TEMPLATE, AND ADT

Polymorphism consists of two words: ***poly,*** which means "***many,***" and ***morph,*** which means "***form***." In programming, polymorphism is the ability to have many forms for the same name. An example of polymorphism is when different functions do a similar task and have the same function name or similarly when using many forms for the same operator. Polymorphism, encapsulation, and inheritance are three important features of object-oriented programming. Function overloading and operator overloading are two examples of polymorphism. A user function or a built-in operator can be overloaded while performing what it was originally designated to do. A computer teacher can become overloaded by teaching an additional math or other course. One form of polymorphism is function overloading. For example, the same function name: ***print( )*** prints the list of items in a queue or a stack, regardless if they were created statically or dynamically. Another example of polymorphism is operator overloading which adds a new meaning to most of the C++ operators. C++ operators are defined for the basic data types; with operation overloading, the same operators can be applied to user-defined types, keeping the operation in the same form as it is applied to the basic types. The goal of operator overloading is to allow user defined types (classes) to behave in the same way as built-in types. Polymorphism contributes to the concept of abstraction by providing a meaningful name that can be used differently for several similar functions and operations. Therefore, a programmer can focus on the function's and operator's work rather than getting involved in the overwhelming details of the implementation. As long as the same interface is exposed, the implementation is irrelevant. Finally, for a language to be considered an object-programming language, it must support the features of polymorphism. C++ templates allow programmers to code and reuse code regardless of the specific data type. When the type of data is defined, the compiler will generate the code as if it was written for that specific data type.

FUNCTION OVERLOADING

A function is overloaded when the same function name is used to perform different tasks depending on the data type. Function overloading is one of the simplest forms of polymorphism. In the following program, there are three versions of **findsmaller**( ); each version of the function takes and returns a different data type. The function **findsmaller**( ) is referred to as an overloaded function. Without the use of function overloading, the same program would require three different function names to perform the same task.

```
#include<iostream.h>
int findsmaller(int x, int y){ if(x<y) return x;  else return y; }
double findsmaller(double x, double y){ if(x<y) return x; else return y; }
char findsmaller(char x, char y){ if(x<y) return x; else return y; }
void main(){
        int a,b;   double c,d;   char e,f;
        cout<<"ENTER TWO INTEGER VALUES: ";
        cin>>a>>b;
        cout<<"ENTER TWO DOUBLE VALUES: ";
        cin>>c>>d;
        cout<<"ENTER TWO CHARACTER VALUES: ";
        cin>>e>>f;
        cout<<"THE SMALLER OF TWO INTEGERS IS: "<<findsmaller(a,b)<<endl;
        cout<<"THE SMALLER OF TWO DOUBLES IS: "<<findsmaller(c,d)<<endl;
        cout<<"THE SMALLER OF TWO CHARACTERS IS: "<<findsmaller(e,f)<<endl;
        }//MAIN
```

Figure 15-1a – Example of function overloading

```
ENTER TWO INTEGER VALUES: 9 10
ENTER TWO DOUBLE VALUES: 5.50 7.75
ENTER TWO CHARACTER VALUES: a z
THE SMALLER OF TWO INTEGERS IS: 9
THE SMALLER OF TWO DOUBLES IS: 5.5
THE SMALLER OF TWO CHARACTERS IS: a
```

Figure 15.1b – Output of 15.1a

HOW DOES FUNCTION OVERLOADING WORK: FUNCTION SIGNATURE

How does the compiler determine which function to call when there are several functions with the same name? A C++ complier uses the function's signature to determine which function to call. A function's signature consists of the function name and the parameter (argument) list. Note that in the function's signature, the return type is not needed because it is necessary in the function's prototype. If there is not an exact match in the argument list, the C++ compiler tries to use a standard type conversion such as converting an integer to a double if necessary. The following program swaps different data types; the compiler is able to figure out the correct function call by using the actual types from the arguments. The number of the arguments and their type may vary in each function.

```
#include<iostream.h>
#include<string.h>
void swap(char * str1, char * str2){
        char temp[80];
        strcpy(temp,str1);
        strcpy (str1,str2);
        strcpy(str2,temp); }//SWAP
void swap(double x, double y){
        double temp;
        temp=x;
        x=y;
        y=temp; }//SWAP
void main(){
        char password[30],newpassword[30];
        double x,y;
        cout<<"ENTER YOUR PASSWORD: ";  cin>>password;
        cout<<"ENTER YOUR NEW PASSWORD: "; cin>>newpassword;
        swap(password,newpassword);
        cout<<"ENTER A VALUE: ";  cin>>x;
        cout<<"ENTER THE NEW VALUE: ";  cin>>y;
        swap(x,y);
        cout<<"YOUR PASSWORD IS: "<<password<<endl;
        cout<<"THE VALUE IS: "<<y<<endl;
        }//MAIN
```

Figure 15.2a – Program using overloaded swap function

```
ENTER YOUR PASSWORD: windows
ENTER YOUR NEW PASSWORD: linux
ENTER A VALUE: 500
ENTER THE NEW VALUE: 1000
YOUR PASSWORD IS: linux
THE VALUE IS: 1000
```

Figure 15.2b – Output of 15.2a

DEFAULT PARAMETER

In a function, a default value for a parameter can be assigned if the function call does not provide one. Default parameters are listed last in the parameter list. When one parameter is assigned a default value, the following parameters should be set to their default values as well otherwise there will be a problem. In the following main program, the function call to **temporary.findgrosspay(1234);** only passes one parameter, the **id**; therefore, the default values will be set for **hoursworked** and **hourlywage**. Default parameters can be very useful in constructors to initialize member data.

```
#include<iostream.h>
class employee{
private: int employeeid, hoursworked;
         double hourlywage, grosspay;
public:  void findgrosspay(int id, int hoursworked=35, double hourlywage=25.00){
                  employeeid=id;
                  grosspay=hoursworked * hourlywage;
                  cout<<"GROSSPAY FOR EMPLOYEE #"<<employeeid;
                  cout<<" is: "<<grosspay<<endl;   }//FINDGROSSPAY
         };//EMPLOYEE
void main(){
         employee temporary;
         temporary.findgrosspay(1234);   }//MAIN
```

Figure 15.3a – Program using default parameters

```
GROSSPAY FOR EMPLOYEE #1234 is: 875
```

Figure 15.3b – Output of 15.3a

OPERATOR OVERLOADING

Defining more than one operation for an existing operator is called ***operator overloading.*** In C++, there are a variety of operators: arithmetic operators such as **+, -, * , /** and **%**, bit wise operators such as **& | ^ , <<,** and **>>**, etc.. With the exception of a few operators, most of the C++ operators can be overloaded. The C++ compiler has already overloaded some of its operators, such as operator + to add an integer as well as a double; and **<<** bit wise left shift operator for the insertion operator (output). In addition, a C++ programmer can overload an existing operator and give it a new meaning. Operator overloading can extend the C++ language and an overloaded operator can be used to overload other operators. The purpose of overloading operators is to make the program easy to read and to apply an operation naturally. In reality, people do not have a problem understanding conventional operators because they are self-explanatory; therefore, the task is to mimic conventional operators and with this in mind, the new overloaded operator should be consistent with the original purpose of the operator.

OPERATOR OVERLOADING EXAMPLES

For example, the operator plus + that adds integers and floating decimals can be overloaded to concatenate two strings. A benefit of overloading is that it makes programming code natural and readable. In the following example, the + operator is overloaded to concatenate the two strings BUTTER and FLY.

string3 = string1 + string2;

Also, without overloading, the concatenation of two strings can be done as follows:

string3= concatenate (string1, string2);

Note that you have to write the function for **concatenate( );**

Another example of the overload operator is the multiplication of two arrays such as matrix multiplication or finding the intersection of two sets. In the statement A=B * C; where A, B and C are matrixes, the multiplication operator * is overloaded for matrix multiplication and the assignment operator = is overloaded to assign the result of the matrix multiplication.

OPERATORS THAT CANNOT BE OVERLOADED

The following C++ operators cannot be overloaded:

?: Conditional expression operator such as **x>y?max=x :max =y;**
. Class (structure) member operators such as **employee.name.**
.* Pointer to member operator such as **employee.*name.**
:: Scope resolution operator such as **stack::top;**
sizeof( ) to determine the size of a type such as **sizeof(int);**

OVERLOAD RESOLUTION RULE

How does the compiler know which function or operator to use now that several exist? The C++ compiler applies a set of rules such as matching the exact type from the argument list or the type of operands. A resolution rule may have to apply a conversion rule. It is possible that one function call (overloaded) matches more than one function's definition; this may cause an ambiguity that may lead to a problem for the compiler that should have been handled properly.

Also, when you overload an operator you cannot change the order of the operator's precedence and you cannot change operator's associativity. In the following example, a- b* c + d, the order of precedence for multiplication is higher than addition, and subtraction is performed first before addition due to its associatively (left to right). Note that when two operators have the same precedence, they work according to their associatively: either from left to right or from right to left.

HOW TO OVERLOAD AN OPERATOR

In order to overload an operator you must write a function for that operator so that C++ will call the function to do the process when the operator is used. The overload function definition is the same as ordinary functions except that the keyword **operator** is used before the parenthesis of the function's argument list. One thing you must realize when you are making an overloaded operator is that you are calling a function except that you do it with a class object. Ask yourself what kind of parameters the function needs and what the function will return. Obviously the overloaded operator will be used in the same

manner as the ordinary operator is used. The following illustrates the general syntax of an overloaded operator prototype with one sample example:

returntype **operator** op (parameters);

OVERLOADING MATH OPERATORS: BINARY + OPERATOR

The binary operator + is used to add two numbers whether they are integers, floats, doubles, or even characters. In fact the C++ compiler has overloaded all these operations under one operator +. The question is how to add more operations to existing operators. C++ allows overloading by simply writing the function for what the overloaded operator should do. After the function is written, the operator will be used in the same way it has been used normally. For example, an employee may have two sources of income, one from a full time job and one from a part time job. Therefore to compute the total salary we have to compute the full time as well as part time salary. Overloading allows the use of the operator + as it is used in the normal addition of numbers, but here + is applied to a user data type which results in self-explanatory programming code.

totalsal= fulltimesal + parttimesal;

Let's not forget that we have to write the function to take care of the above overloading. Once the function is written, then naturally the operator can be used over and over. If we did not want to use overloading, our statement would look like the following, possibly with several parameters.

totsalary = findfulltimesal(.....) + findparttimesal(....);

OVERLOADING BINARY + OPERATOR: PROGRAM

The following program illustrates overloading the + operator to add two objects of the **employee** class known as **fulltime** and **parttime**. This allows us to calculate the total salary of an employee whose income comes from two different sources.

```
#include<iostream.h>
class employee{
private: long int id;
         double salary;
public:  employee(const long int empid, const double empsal);
         double operator+(const employee &emp); };
employee::employee(const long int empid, const double empsal){
         id=empid;
         salary=empsal; }//CONSTRUCTOR
double employee::operator+(const employee &emp){
         double totsalary;
         totsalary= salary + emp.salary;
         return totsalary; }//OP+
void main(){
         employee fulltime(1234599,70000.00), parttime(1234599, 10000.00);
         double totsalary=fulltime+parttime;
         cout<<"Total Salary: "<<totsalary;  }//MAIN
```

Figure 15.4a – Example of overloaded binary operator

```
Total Salary: 80000
```

Figure 15.4b – Output of 15.4a

PREFIX AND POSTFIX OPERATOR OVERLOADING

Prefix or postfix operators, ++ (increment operator) and -- (decrement operator), can be overloaded for use with user defined types in a similar way as other operators. Again the idea of overloading is to make the program more understandable and not to make it confusing. You do not want to overload ++ to decrement or overload -- to perform increment. The following simple program illustrates the overloading of the increment operator ++ for a type known as a date. The program keeps track of the working day by adding one to each day with a range of 1 to 5. The same program can be extended to add one to the day of the month.

The general form of an increment overloading is:

classtype &operator++(classtype &obj)

```
#include<iostream.h>
class date{
public:
        date(){ wday=1; }
        int getwday(){ return wday;}
        void operator ++ ( ){
                if(wday>5) wday=1;
              else ++wday;}
private:
        int wday; };//DATE

void main(){
        date workday;
        ++workday;
        cout<<"WORK DAY IS: "<< workday.getwday()<<endl;
}//MAIN
```

Figure 15.5a – Example of increment operator overloading

```
WORK DAY IS: 2
```

Figure 15.5b – Output of 15.5a

FRIEND FUNCTION

A friend function is not a member of a class but it has access to the class's private and protected members. A friend function is declared in the class that grants the access. To recall every member function implicitly has a pointer to its object known as **this** pointer. However a friend function does not have a **this** pointer and for this reason all of the necessary objects need to be passed explicitly to the friend function. For example, using a friend function to overload a binary operator. To declare a function as a friend of a class, you must precede the function prototype in the class definition with the keyword friend.

WHAT IS this POINTER?

Every member function has an implied parameter, called **this** pointer. The value of **this** points to the class-object (class argument) of the member function. In order to refer to the implicit argument that is passed to the member function, the keyword **this** preceded by an asterisk is used.

The following three statements perform the same task and show how **this** pointer is used within a member function. In the first version the keyword **this** is implied.

```
print( ){ cout<<findgrosspay(); }
print ( ){cout <<(*this).findgrosspay();
print( ){cout<<this->findgrosspay();
```

Friend functions and **this** pointers are further discussed in a later chapter.

OVERLOADI NGINPUT AND OUTPUT OPERATORS: >> AND <<

In C language the operators >> and << are used as right-shift and left-shift, respectively. However in C++ these operators are overloaded for input (extraction) and output (insertion). You can also overload these operators to work with your own abstract data type (class object). The function that does the overloading **operator <<( )** is a friend function since it is not a class member. This function returns a reference to **ostream**; this is done by placing an **&** in the function header. The two arguments are a reference to an **ostream** object and a reference to the object that is to be output.

The general prototypes for >> and << overloading can be shown as follows:

```
istream& operator >> (istream& parameter1, datatype&  parameter2);
ostream& operator <<(ostream& parameter1, datatype& parameter2);
```

EXAMPLE OF I/O OVERLOADING

How is it possible to input or output a long list of information for an employee class in one instance without going through field by field? One solution is to overload the input or output operators so that when they are used in the program they can input and output all the data. In fact, you can customize the output the way you want to and for beginners I/O overloading is a meaningful example of how overloading can be useful. In the following example the >> extraction and << insertion operators have been overloaded.

```
#include<iostream.h>
#include<string.h>
class employee{
private: char name[20];
          int hw;
          double hr,taxrate;
          double grosspay,taxamount,netpay;
public: void compute(){
          taxrate=0.15;
          grosspay=hw*hr;
          taxamount=grosspay*taxrate;
          netpay=grosspay-taxamount; }//COMPUTE
          friend  istream& operator >>(istream& cin , employee& individual );
          friend ostream&  operator << (ostream& cout, const employee& individual);
          };//EMPLOYEE CLASS
istream& operator >>(istream& cin , employee& individual){
          cout<<"ENTER NAME, HOURS WORKED, HOURLY RATE: ";
          cin>>individual.name>>individual.hw>>individual.hr;
          return cin; }// >>
ostream& operator <<(ostream& cout, const employee& individual){
          cout<<"EMPLOYEE NAME: "<<individual.name<<endl;
          cout<<"GROSS PAY: $"<<individual.grosspay<<endl;
          cout<<"TAX AMOUT: $"<<individual.taxamount<<endl;
          cout<<"NET PAY: $"<<individual.netpay<<endl;
          return cout; }//<<
void main(){
          employee regularemp;
          cin>>regularemp;
          regularemp.compute();
          cout<<regularemp;
          }//MAIN
```

Figure 15.6a – Example of I/O overloading

```
ENTER NAME, HOURS WORKED, HOURLY RATE: Ebrahimi 45 18
EMPLOYEE NAME: Ebrahimi
GROSS PAY: $810
TAX AMOUT: $121.5
NET PAY: $688.5
```

Figure 15.6b – Output of 15.6a

OVERLOADING ASSIGNMENT OPERATOR =

The operator = is used for assignment meaning the value of the right hand side is placed into the content on the left hand side. In the following example: int x,y=5; x=y; the assignment places 5 in x. What happens when you assign one object to another object? For example, in: date x,y; x=y; the object y is copied to object x and the memory is overwritten. An object may have more than one value and the overloaded assignment operator will assign all the values of the object on the right hand side to the values of the object on the left hand side.

The general form of the = overloading is shown as follows:

```
classtype &operator=(classtype &object);
classtype x,y;
x=y;
```

The above function takes a reference to the object and returns a reference from the object. An example where an overloaded assignment operator could be used is to copy an employee's information such as name, hours worked, and hourly wage into another object such as:

```
class employee{   string name;
                  double hoursworked, hourlywage;

                  }//EMPLOYEE
employee manager, ebrahimi;
manager=ebrahimi;
```

OVERLOADING THE EQUALITY OPERATOR = =

The equality operator = = can be overloaded so that it can be applied to class objects, for example comparing the time of two objects known as time1 and time2. Note that after the = = operator is overloaded the following statement could be used:

```
if (time1= =time2) cout<<" BOTH TIME ARE EQUAL";
```

To overload the = = operator we have to write a function that takes an argument of the class type and compares its data members with the data members of the passed object (right side) and returns a Boolean value (true or false). The following is the function to overload the equality operator for the time class; it compares two time objects to see if they are equal by testing the hours, minutes, and seconds.

```
bool time::operator= =(const time& appointment){
if ((hour    appointment.hour)&&(minute    appointment.minute) &&
                                   (second==appointment.second))  return true;
else return false;  }// END= =
```

OVERLOADING AN INDEX OPERATOR [ ]

The index or subscript operator is used to index arrays; an integer number ranging from zero to one less than the maximum size of the array indicates the position. By overloading an index operator, the index can be a basic data type such as integer or a user defined type (class objects). The overloaded index operator is not restricted to work with arrays only but with any other collection of items (containers). By overloading [ ], we can mimic the linked list to work like an array. One example of index overloading is to find the frequency of an object's occurrence, such as the frequency of each word in a document. The function to handle the index operator takes two parameters: the first is a reference to an object of the class that is going to be used as an array, and the second argument may be of any type such as an integer or class of an object to be used as the index.

```
int & operator[](const  str& s);
```

Note that in the following statement that uses overloaded [ ], the string s is used as an index.

```
freq[s]=freq[s]+1;
```

OVERLOADING ISSUES AND LIMITATIONS

Most of the C++ operators can be overloaded, but there are few that cannot be overloaded. Keep in mind, operator overloading should make the program easier and the programmer should be able to use the operator in a natural way and eventually overloading should lead to reusability and less programming code. Improper usage of operator overloading can result in programming chaos. For example, overloading ++ to perform subtraction instead of addition would lead to confusion. Depending on what you want to do, you may want to overload an operator. Among the more frequently used overloaded operators we can name +, >>, <<, =, = =, and ++. Notice that while you can add additional meaning to the existing operators, you cannot create a new operator even though it is highly desirable such as exponentiation (**). Moreover, you cannot change the required number of operands; for example, a unary operator cannot have more than one operand.

CONSTRUCTOR OVERLOADING

A constructor is a member class that initializes an object and is called automatically as the object is created. Remember, a constructor has the same name as its class and does not return a value. Moreover, a constructor that has no parameters is known as a default constructor.

A constructor can be overloaded with different data types. For example, a stack class can have more than one constructor to initialize the stack with character type numbers or to initialize the stack with integer type numbers.

BINARY + OPERATOR OVERLOADING: TIME CLASS PROGRAM

The following program overloads the binary operator + for adding two objects of the class **time** known as **begintime** and **durationtime** and assigns the result into a third object, **finishtime**. Keep in mind, the overloaded + operator adds the elements of objects, not two values of basic data types such as integer.

```
#include<iostream.h>
class time{
private: int hour, minute, second;
public: time(){ hour=0; minute=0; second=0; }
        time(int hr, int mn, int sec){
                hour=hr; minute=mn; second=sec; }
time operator+(const time& t){
        time totaltime;
        totaltime.hour = hour + t.hour;
        totaltime.minute = minute + t.minute;
        totaltime.second = second + t.second;
        return totaltime; }//+
friend ostream& operator<<(ostream&, const time&); };//TIME
ostream& operator<<(ostream& cout, const time& t){
        cout<<t.hour<<":"<<t.minute<<":"<<t.second<<endl;
        return cout; }//<<
void main(){
        int hr, mn, sec;
        cout<<"ENTER HOUR, MINUTE, AND SECOND: ";
        cin>>hr>>mn>>sec;
        time begintime(hr,mn,sec);
        cout<<"ENTER HOUR, MINUTE, AND SECOND: ";
        cin>>hr>>mn>>sec;
        time  durationtime(hr,mn,sec);
        time finishtime= begintime + durationtime;
        cout<<"FINISH TIME: "<<finishtime;   }//MAIN
```

Figure 15.7a – Example of overloading the + operator

```
ENTER HOUR, MINUTE, AND SECOND: 1 15 10
ENTER HOUR, MINUTE, AND SECOND: 1 11 20
FINISH TIME: 2:26:30
```

Figure 15.7b – Output of 15.7a

ABSTRACTION

Abstraction is the process of focusing on the essential and relevant aspects of an object or a task (function) without getting bogged down in their unnecessary details (molecular structure). Abstraction means to generalize an object and hide its details. To apply abstraction to objects, one must classify the objects according to their similarities as to what they do and what they have. One method of abstraction is to classify objects according to the domain of the problem as to the requirements at the moment, the characteristics of the objects, and how these objects interact with each other. An abstraction can have many layers of abstraction itself. Abstraction is around us daily, we express and represent ideas in abstract terms. We should not confuse abstraction with impossibilities (e.g. painting); however, some abstractions are difficult to figure out depending on the situation. You can think of abstraction as a box, and within the box is another box, and so on. How would you use abstraction to classify the following objects? Lion, tiger, cat, mouse, horse, rooster, bat, fish, frog, lobster, snake. Can you categorize the objects around you in any of the four elements of air, earth, fire, and water?

DIFFERENT LAYERS OF ABSTRACTION

The world around us is made of objects and each object has a name that represents an abstraction. For example: car, book, and computer each represent an abstract concept that can each be overwhelmingly described. An object or a thought can be represented in many layers of other objects and thoughts. Depending on what is important at each stage, the detail is either hidden or revealed. To understand what abstraction is and how it is used, look at the following example. A computer is an abstract object which itself is made of other abstract objects such as a mouse, keyboard, CPU, memory, and programs. A program is an abstract object that is made of other programs or statements. Similarly, a statement is made of identifiers (names) that may each represent a memory and the operations that are applied. Each operation is made of instructions and each instruction is made of microinstructions. Can you imagine if we reveal all the above interactions each time a user types a word? It is overwhelming and undesirable. Let us not forget that while the details of how a word is stored and how it interacts with other words is not important to a word processor, that same information is crucial for a compiler writer or a computer architect. What we want to do is classify each concept and wrap it (encapsulate) like a box under a name and provide an interface to deal with the box when it is necessary. To wrap it up, abstraction allows a programmer to look at the big picture and to focus only on important parts of problem that need to be solved. You should realize that there are two parts to an abstraction: interface and implementation. For example, a computer itself is an abstraction and when we are using it through an interface we are not concern about its detail. In programming, the abstraction concept is everywhere and has evolved into two main categories from procedural to data abstraction. For example, use of functions in the main program is the procedural abstraction and use of objects is a data abstraction. An example of procedural abstraction would be a function called **sort( )** and an example of data abstraction would be a class called **employee**. An abstract data type (ADT) is a

higher level of abstraction than what C++ data types provide. Some important examples of abstract data types are stacks, queues, and trees.

ABSTRACT DATA TYPE (ADT)

The concepts of hiding detail (information hiding) and encapsulation are related to abstract data types. ADT emphasizes the work that is to be done rather than how it is done. The term ADT refers to the classes with information hiding that can be implemented by a programmer and then used by another programmer without knowing the details of how these classes have been implemented. Note again, the programmer that uses the class is not concerned (doesn't care) with how the class is implemented, but what the class is going to do. Therefore any detailed information can have a negative impact and can be misused. For experienced programmers terms such as stack, queue, tree, and graph have special meaning. In fact these terms are ADTs, each with its own data and operations. When these ADTs are implemented, they are ready to be used and the user should focus on how to use them rather than getting involved in the details of how they are designed. An important step in software engineering is creating an efficient ADT initially; this ADT becomes a pioneer tool for further expansion of the program. ADTs allow us to separate the interface (program interaction) from any particular data structure and algorithm implementation.

CLASS AND OBJECT

The world around us is composed of objects like book, mouse, and window. Each object comes from a class or blueprint (pattern). Classes correspond to categories (classifications) and are general terms while objects correspond to individuals (instances) and what you are dealing with at the moment. You can say a class is a blueprint and an object is an instance of the class. Think of a house blueprint and your own house. The kind of house is the class and your particular house is an object. Every object is an instance of a class. For example, date is a class and my birthday is an instance of class date.

INFORMATION HIDING (DATA HIDING)

Information hiding is applying an access restriction on data that the user should not be exposed to. Data hiding helps prevent errors that result from intentional or inadvertent changes in data. Keep in mind that by applying abstraction we are hiding and encapsulating data and vise versa. Think of your computer as an object of a computer class where internal components (details) are hidden in a box and communication is done through an interface (keyboard, mouse, or screen).

MODULE

A group of several related functions with the data they manipulate is known as a module. In a modular programming, a program (large program) is divided into several modules where in each module its data is hidden. Each module is placed into separate namespaces or files, which can also be compiled separately.

How the module works (user interface) is kept separate from how the module is implemented. Separating the user interface from the implementation enables a user to change the implementation without changing the interface.

HOW TO MAKE AN INCLUDE FILE

By this time you are fully aware that every C++ program requires an include file ranging from I/O to math and standard libraries. In fact one reason the C/C++ language is a compact, and as result, fast language is that you only include what you need in a program. Even **cin** and **cout** are not part of the C++ language itself, you have to include the library for them. For example, if you need to do a variety of mathematical functions, you place **#include <math.h>** in your program. The C++ compiler keeps the program and data for these libraries in separate files (modules) and when needed you can add them to the head (top) of your program. Do you know why these files are called header files? Using these modules contributes to information hiding and a programmer can make their own header files and include them in a program when needed. Commonly the class and function prototypes are kept in a header file. To make a header file, just as making any ordinary file, type the information and save it under a filename with extension **.h** (for example **employee.h**). To use your own include file, instead of angle brackets, surround the file name in double quotations as shown below.

```
#include "employee.h"
```

ENCAPSULATION

Encapsulation is a way to put together (to enclose in a capsule) data (attributes) and the functions (behavior) that manipulate the data into one self-contained unit (package) and safeguard it from outside misuse. In object-oriented programming the encapsulation concept is achieved by means of a class where data and functions can be either private or public to the object. The private data members and member functions are hidden from the class user; they are encapsulated within the class and they are not accessible from outside of the class. However, through the public parts (as an interface) of a class, the private elements can be accessed.

HOW ABSTRACT DATA TYPES (ADT) ARE IMPLEMENTED

In C++, abstract data types (ADT) are implemented by having a class where the member variables are declared as private and some of the member functions are public. These public member functions are used as an interface to access the private members. Putting an access restriction on the data in a class will maintain data integrity and consistency. The definition of class can be placed in a separate file and then be compiled; and whenever it is needed it can be included in different programs. Similarly the class implementation can be placed in a different file that can be included with other programs that need them.

TEMPLATE

How would you write a program that searches for a number like salary or for an employee's name? How would you write a stack program where you can push either a number or a string? You may respond quickly that you would write a separate program for each data type. By using **template**, a C++ programmer can design generic algorithms and generic classes. With a template your algorithm can work for integers as well as floating-point numbers. In C++, you specify what type of data to work with by the function call or when an object is declared (instantiated). One advantage of the template is that it promotes program reusability and in the long run programming becomes easier.

TEMPLATE FUNCTION

A template function is a function that works with a generic data type such as writing one function that works with any type of data, whether it is a built-in or user-defined type. In a template function the actual data type is not specified; rather, the generic data type is used. On a function call with the arguments (actual data types), the compiler will substitute the parameters for the actual data type such as: integer, character, or other data type. The actual value type parameter can also be used for local variables as well as the return value of the function. A template function includes a template definition followed by a function definition. The function template starts with the keyword template following by an angle bracket enclosing the keyword class with a parameterized data type (generic type) name.

After the template definition the function definition would be the same except the generic type name would be used for the parameterized data type. The general syntax of a template function is shown below:

```
template < class generictypename>
generictypename functionname (generictypename parametername){
        functionbody;
        return functionvalue; }
```

TEMPLATE FUNCTION: PROGRAM

The following program finds the smaller of two values whether they are integers, characters, or floating decimal numbers. The template function will take care of the different data types. The generic data type in the template function will be substituted with the actual data type provided by the calling function. Note that GT is an identifier name and as a convention I used capital letters to represent it. Other conventions have used identifier names such as T and Type.

```
#include<iostream.h>
template <class GT>
GT smaller (GT x, GT y){
        if (x < y) return x;
        else return y; }//SMALLER
void main(){
        int i,j;
        cout<<"ENTER TWO INTEGERS: ";
        cin>>i>>j;
        cout<<"THE SMALLER OF THE TWO IS: "<<smaller(i,j)<<endl;
        char  c, d;
        cout<<"ENTER TWO CHARACTERS: ";
        cin>>c>>d;
        cout<<"THE SMALLER OF THE TWO IS: "<<smaller(c,d)<<endl;
        float f,g;
        cout<<"ENTER TWO FLOATING POINT NUMBERS: ";
        cin>>f>>g;
        cout<<"THE SMALLER OF THE TWO IS: "<<smaller(f,g)<<endl;
        }//MAIN
```

Figure 15.8a – Example of a template function

```
ENTER TWO INTEGERS: 10 5
THE SMALLER OF THE TWO IS: 5
ENTER TWO CHARACTERS: a c
THE SMALLER OF THE TWO IS: a
ENTER TWO FLOATING POINT NUMBERS: 2.75 12.50
THE SMALLER OF THE TWO IS: 2.75
```

Figure 15.8b – Output of 15.8a

TEMPLATE WITH MULTIPLE TYPES

It is possible for a template to have more than one type. For instance, in computing gross pay, the hours worked is an integer, however the hourly rate and gross pay are doubles. The general syntax of a template function having multiple types is as follows:

```
template <class T1, class T2>
T2  computegrosspay (T1 hoursworked,  T2 hourlyrate){
                T2  grosspay;
                 grosspay=hoursworked * hourlyrate; return grosspay; }//GROSS

T2 computetotalprice (T1  quantity, T2 unitprice){
               T2 totalprice;
                totalprice=quantity * unitprice; }//TOTALPRICE
```

In the above two examples, T1 and T2 are the type identifiers that will be replaced by the two different actual data types.

OVERLOADED FUNCTIONS VERSUS TEMPLATE FUNCTIONS

Function overloading becomes useful when performing similar operations on different data types. In function overloading we use only one function name with several different versions of the function even though the whole operation is identical. However when using template functions, we have one function name with one version of the function that is parameterized and the compiler generates different versions of the function based on the data type provided by the function call. For example we could have a function called sort to be overloaded so it will cover a variety of sorting algorithms (generic algorithms) with many different swaps. One disadvantage of function overloading is that if one function needs to be changed, all the functions must be changed. It should be noted that a template function could be overloaded in the same way as a regular function.

CLASS TEMPLATE

A class template is a class that can have a generic data type instead of an actual data type. The actual data type for a class will be provided as the class is instantiated and the compiler will generate the proper code by substituting the generic type with the actual type.

The general syntax for a class template is the same as a function template, which starts with the keyword template followed by the keyword class and its parameterized type enclosed in angle brackets.

The class definition is done in the same way as an ordinary class, except that the generic type (parameterized type) is used in place of an actual type. Remember to place the actual data type in angle brackets when the object is instantiated. One good usage of class

templates is to build a generic stack, queue, or tree. You can define a generic stack with an unspecified data type. Remember that every time you use a template class or its member functions, the template definition must precede the class definition or function definition. For clarity you may want to place the template definition on a separate line.

```
#include<iostream.h>
template <class T>
class employee{
private: T hoursworked, hourlyrate, grosspay;
public: employee (T hw, T hr ){ hoursworked=hw; hourlyrate=hr; }
        void resetdata( T hw, T hr ){ hoursworked=hw; hourlyrate=hr; }
        T computegrosspay( ){ grosspay= hoursworked * hourlyrate; return grosspay; }
 }; //EMPLOYEE
void main(){
        employee <double> manager(40.5, 100.0);
        cout<<"SALARY IS "<<manager.computegrosspay()<<endl;
        manager.resetdata(30.0, 50.0);
        cout<<"SALARY IS "<<manager.computegrosspay()<<endl;    }//MAIN
```

Figure 15.9a – Example of a class template

```
SALARY IS 4050
SALARY IS 1500
```

Figure 15.9b – Output of 15.9a

CLASS TEMPLATE: GENERAL SYNTAX

When you are using a class template, make sure to distinguish between the general syntax for the class itself, its members, and when the object is instantiated. The following example demonstrates each of the above cases:

```
template <class typeidentifier>
class classname{

        typeidentifier ...variablename.
        typeidentifier functionname (typeidentifier....);
        }; //CLASS

template <class typeidentifier>
typeidentifier classname <typeidentifier>:: functionname (typeidentifier...){

        return ....; }//FUNCTION
main(){

        classname <actualtype> objectname
        ..... }//MAIN
```

STACK WITH TEMPLATE: GENERIC CLASS PROGRAM

A stack is a data structure and an ADT where items are added and removed from one end. The function **push**( ) inserts a new item to the stack and **pop**( ) removes the item that was most recently inserted. To summarize the stack ADT, items are removed according to the last-in-first-out discipline. The functions **isempty**( ) and **isfull**( ) check the stack to determine whether it is empty or full, respectively. The following program defines a generic class called **stack** with a maximum of 5 items, including double and character values. The operand stack is used to hold the numbers while the operator stack holds the character operators. The stack can be extended to hold other different data types as well.

```
#include <iostream.h>
const int MAXSIZE=5;
template <class T>
class stack{
public: stack(void);
        bool isempty() const;
        bool isfull() const;
        void push(T x);
        T pop();
private: int top;
         T items[MAXSIZE];   };  //STACK
template<class T>
stack< T > :: stack(){  top = -1;}
template<class T>
bool stack <T> :: isempty() const{ return(top == -1); }//ISEMPTY
template<class T>
bool stack <T> :: isfull() const{ return(top == (MAXSIZE-1));  }//ISFULL
template<class T>
void stack<T> :: push (T x){
       if (isfull())  cout<<"YOU CAN'T PUSH"<<endl;
      else { top++; items [top] = x; } }//PUSH
template<class T>
T stack <T>:: pop(){
     if (isempty()){cout<<"YOU CAN'T POP"<<endl;   return 0; }//IF
     else return items [top--];  }//POP
void main( ){
     stack<double> operandstack;
     operandstack.push(3.00);   operandstack.push(12.5);    operandstack.push(1.2);
     cout<<"STACK VALUE IS:"<< operandstack.pop()<<endl;
     operandstack.push(8.00);   operandstack.push(2.00);    operandstack.push(5.4);
     while (! operandstack.isempty()) cout<<"STACK VALUE IS:"<< operandstack.pop()<<endl;
     stack <char> operatorstack;
     operatorstack.push('+'); operatorstack.push('*');
     operatorstack.push('-');operatorstack.push('/');
     while (! operatorstack.isempty()) cout<<"STACK VALUE IS:"<< operatorstack.pop()<<" "<<endl;
}//MAIN
```

Figure 15.10a – Stack template program

```
STACK VALUE IS:1.2
STACK VALUE IS:5.4
STACK VALUE IS:2
STACK VALUE IS:8
STACK VALUE IS:12.5
STACK VALUE IS:3
STACK VALUE IS:/
STACK VALUE IS:-
STACK VALUE IS:*
STACK VALUE IS:+
```

Figure 15.10b – Output of 15.10a

QUEUE WITH TEMPLATE

A queue is a data structure or an abstract data type (ADT) where insertion is done on one end and removal is done on the other end. The following example implements a queue with a template, enabling the queue to work with a variety of data types. The following template queue is tested with **int** and **char** types, however one can extend the template queue to work with other data types and include overloading, making it more powerful and complex.

```
#include <iostream.h>
const int MAXSIZE = 10;
template <class T>
class queue {
private: int front, rear,size;
         T info[MAXSIZE];
public:  queue(int);
         void enqueue(T x);
         T dequeue( );
         bool isempty( ) const;
         bool isfull() const; };//CLASS QUEUE
template<class T>
queue <T>:: queue(int s=MAXSIZE){
          front = 0;
          rear = 0;
          size=s+1;   }//CONSTRUCTOR
template<class T>
bool queue <T>:: isempty() const{  return (rear == front); }
template<class T>
bool  queue <T>:: isfull() const {   return ((rear + 1) % size == front);  }//ISFULL
template<class T>
void queue <T>::enqueue(T  x){
          if(isfull()) cout<<"QUEUE FULL:"<<endl;
          else{  rear = (rear +1) % size;
```

```
                info[rear] = x; }        }//ENQEUE
template<class T>
T queue <T>::dequeue( ){
        if( isempty() ){cout<<"QUEUE EMPTY"<<endl;
                        return -1;}//IF
        else{ front = (front + 1) % size;
                return info[front];}//ELSE
}//ENQUEUE
void main(){
    queue <int> myintqueue(5);
    for (int i=0;  i<=5; i++){
    myintqueue.enqueue(i); }//FOR
    while(!myintqueue.isempty()){ cout<<myintqueue.dequeue()<<" "<<endl;}//WHILE
    queue <char> mycharqueue(4);
    for (char c='A';  c<='E'; c++){
    myintqueue.enqueue(c); }//FOR
    while(!myintqueue.isempty()){ cout<<char(myintqueue.dequeue())<<" "<<endl;}//WHILE
  }///MAIN
```

Figure 15.11a – Queue template example

```
QUEUE FULL:
0
1
2
3
4
A
B
C
D
E
```

Figure 15.11b – Output of 15.11a

TOPICS NOT COVERED HERE

The topic of overloading, especially operator overloading, can become tedious and error prone. When an operator is overloaded, there are many versions of its function and, on top of the ambiguity as to which function to pick, conversion of one type to another must take place. Type conversion by itself needs lengthy coverage as well as how to deal with passing an object as a parameter or returning an object from a function, e.g. passing a large object versus small one. Controversial issues surrounding the friend function are not discussed here. Virtual functions need their own attention as to how C++ chooses the correct overridden functions during run-time, creating run-time polymorphism. Virtual functions become a very interesting subject when they work with a pointer to the base class and then each derived class calls its own function although the function name

remains the same. The advantage of templates is that the programmer can design generic algorithms and classes. However, templates have time and space overhead, which may not be ideal for real-time applications. The use of templates can become more complicated by introducing nested templates with multiple parameters. The topic of polymorphism with virtual functions requires knowledge of inheritance and will be discussed in the next chapter.

CLOSING REMARKS AND LOOKING AHEAD

Whether you want to apply brakes on a bike, car, van, or truck, they all have one task in common; the result is the vehicle coming to a stop. Also, the process of finding the area of a geometric shape is similar whether the area is of a rectangle, triangle, or circle. The same is true for printing, regardless of whether you are printing a list, queue, or stack. One benefit of overloading is the ability to give one meaningful name to several functions that have the same task, rather than giving a different name to each function. For example, in a payroll system many functions with the same name, such as payment( ), can be used for hourly paid or salaried employees as well as for consultants and managers. Another example, double-clicking on an icon may provide different results depending on whether the icon is a word processor, e-mail application, or an executable file.

Polymorphism allows an entity (for example a variable, function, or object) to take on a variety of representations. Function overloading allows us to reuse the same name for functions (or methods) as long as the parameter list differs.

Many operations on newly defined user data types share a common ground with the operations on existing basic data types. Therefore, it is desirable that new user data types use the same operator in the same manner. The process of enabling C++ operators to work with objects is called overloading. In operator overloading, a function is written for the operator so that it can be used normally with the class object. One example is to define a complex number data type and use the same operators such as +, -, *, and / to perform the operations on complex numbers as they would perform on a real number. Another example of overloading operator is to apply it to the object of class date, time, and polynomial. With the use of a template, a C++ programmer is able to write functions and classes regardless of their specific data types.

The next chapter will cover the last aspect of object-oriented programming known as inheritance: when one object carries the properties of one or more other objects. Inheritance is one of the key concepts of object-oriented programming where member variables in one class automatically become part of objects belonging to another class. One class is considered the base class, and the class that inherits from the base class is considered the derived class.

Inheritance, in addition to encapsulation (class) and polymorphism, is a key element of object-oriented programming and enhances programming by reusing and sharing code; in the long run, inheritance is cost effective and minimizes programming errors.