Cairo University
Faculty of Computers and Artificial Intelligence
Computer Science Department

Concepts of Programming Languages

Generic Programming

Dr. Amin Allam

In this lecture, we discuss various methods for *generic programming* in *compiled languages*. We discuss their reasoning, benefits, and limitations.

1 Variable parameters

Allowing a *subprogram* to take variable numbers and types of its *parameters* in different calls enhances the *generality* of this *subprogram*, with the drawback of absence of *type checking*.

Consider the following C++ function which takes two integers, `ni` and `nd`, that *should be* followed by `ni` integers and `nd` doubles, and returns the average of all these values:

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

double Avg(int ni, int nd, ...)
{
    int i; double d=0;
    va_list v; // Used to iterate over the variable parameters
    va_start(v, nd); // Start the variable parameters after nd
    for(i=0; i<ni; i++) d+=va_arg(v, int); // Get ni integers
    for(i=0; i<nd; i++) d+=va_arg(v, double); // Get nd doubles
    va_end(v); // Finalize traversal
    return d/(ni+nd);
}
int main()
{
    cout << Avg(2,3, 1,1, 1.0,1.0,1.0) << endl; // 1 (Correct)
    cout << Avg(3,1, 1,2,3, 4.0) << endl; // 2.5 (Correct)
    cout << Avg(1,1, 1, 1.0) << endl; // 1 (Correct)
    cout << Avg(1,1, 1, 1); // 0.5 (Incorrect parameter types and result)
    return 0; // The above error because last integer is interpreted as double!
}
```

The C `printf()` function outputs parameters whose types are specified in the first input string. For example, `printf("%d %lf", 1, 2.0);` prints an integer followed by a double. Compilers actually check the `printf()` parameters *statically* against the input string to avoid the catastrophic *type errors* such as the error in the above `Avg()` function.

2 Subprograms as parameters

Consider a high level function `Copy()` depending on two low level functions as follows:

```
void Copy()
{   while(true)
    {char c=ReadCharFromKeyboard(); WriteCharToPrinter(c);}
}
```

Suppose we need to do the same behaviour, but to copy from a file to screen, instead of copying from keyboard to printer. We must define another function that has very similar behaviour of `Copy()` but depends on different low level functions:

```
void Copy2()
{   while(true)
    {char c=ReadCharFromFile(); WriteCharToScreen(c);}
}
```

This causes duplicating the code of the original `Copy()` and replacing the inside function calls with new function calls. We can avoid this code duplication to improve *writability* by making the `Copy()` function depend upon two *function abstractions* (implemented as *pointers to functions*) instead of two specific low level functions as follows:

```
char ReadCharFromKeyboard() {/*Read char from keyboard*/}
char ReadCharFromFile() {/*Read char from file*/}
void WriteCharToPrinter(char c) {/*Write char to printer*/}
void WriteCharToScreen(char c) {/*Write char to screen*/}

void Copy(char (*ReadChar)(), void (*WriteChar)(char))
{
    while(true) {char c=ReadChar(); WriteChar(c);}
}
void TestCopy()
{
    Copy(ReadCharFromKeyboard, WriteCharToPrinter);
    Copy(ReadCharFromFile, WriteCharToScreen);
}
```

The *calls* to `ReadChar()` and `WriteChar()` *dynamically bind* to *functions* according to the values of the *function pointers* which are known only at *run time*. Thus, the above technique is compatible with the SOLID *dependency inversion principle*:
"High level modules should not depend upon low level modules. Both should depend upon abstractions." ⇒ "Program to an interface, not an implementation."

The `Copy()` function definition needs to change only if the behaviour (or logic) of copying changes. It does not need to change if the reader (source) or writer (destination) changes. Thus, the above technique is also compatible with the SOLID *single responsibility principle*:
"A module should have only one reason to change."

3 Type-less parameters

Consider the following program which sorts an array using the standard C `qsort()` function:

```
#include <stdlib.h>  // includes qsort()

int CompareInt(const void* pa, const void* pb)
{
   int a=*(int*)(pa), b=*(int*)(pb);
   return a-b;   // returns 0 if a==b, +ve if a>b, -ve if a<b
}
int CompareDouble(const void* pa, const void* pb)
{
   double a=*(double*)(pa), b=*(double*)(pb);
   double d=a-b;
   // consider them equal if the difference is very small
   if(d>-0.00001 && d<0.00001) return 0;
   if(d>0) return 1;
   return -1;   // d<0
}

int main()
{
   int a[]={3,5,2,1,7,4,6};
   int n=sizeof(a)/sizeof(a[0]);
   qsort(a, n, sizeof(int), CompareInt);
   for(int i=0;i<n;i++) cout<<a[i]<<" "; cout<<endl;

   double b[]={3.1, 5.3, 2.2, 1.1, 7.4, 4.8, 6.9};
   int m=sizeof(b)/sizeof(b[0]);
   qsort(b, m, sizeof(double), CompareDouble);
   for(int i=0;i<m;i++) cout<<b[i]<<" "; cout<<endl;
   return 0;
}
```

The idea is to use the `void*` data type which accepts any pointer type, with the drawback of absence of *type checking*. The first argument is a `void*` which accepts the array to be sorted. The second argument is the number of elements of the array. The third argument is the size of each element (because the function does not have any information about the type of elements). The fourth argument is a pointer to a function which compares two elements and returns 0 if they are equal, a positive value if the first is larger, and a negative value if the second is larger.

The address of each element is passed as `void*` to match the function declaration and then a *type cast* is performed from within the function according to the specific data type. The following is an implementation of a `SelectionSort()` function with the same declaration of the standard `qsort()` function:

```
#include <stdlib.h>  // includes malloc()
#include <string.h>  // includes memcpy()

void SelectionSort(void* buf, int n, int elem_size,
                   int (*Compare)(const void*, const void*))
{
   int i,j;
   // allocate memory for the temporary variable
   void* ptemp=malloc(elem_size);
   for(i=0;i<n;i++)
   {
      // Keep the minimum of a[i ... j] in a[i]
      for(j=i+1;j<n;j++)
      {
         void* pa=(char*)buf+(i*elem_size);   // pa=&a[i]
         void* pb=(char*)buf+(j*elem_size);   // pb=&a[j]

         if(Compare(pb, pa)<0)   // if(a[j]<a[i])
         {
            memcpy(ptemp, pa, elem_size);   // temp=a[i];
            memcpy(pa, pb, elem_size);      // a[i]=a[j];
            memcpy(pb, ptemp, elem_size);   // a[j]=temp;
         }
      }
      // Here: a[i] is the minimum of a[i ... n]
      // Hence: a [0... i] are sorted
   }
   // de-allocate memory of the temporary variable
   free(ptemp);
}

int main()
{
   int a[]={3,5,2,1,7,4,6};
   int n=sizeof(a)/sizeof(a[0]);
   Qsort(a, n, sizeof(int), CompareInt);
   for(int i=0;i<n;i++) cout<<a[i]<<" "; cout<<endl;

   double b[]={3.1, 5.3, 2.2, 1.1, 7.4, 4.8, 6.9};
   int m=sizeof(b)/sizeof(b[0]);
   Qsort(b, m, sizeof(double), CompareDouble);
   for(int i=0;i<m;i++) cout<<b[i]<<" "; cout<<endl;
   return 0;
}
```

4 Polymorphism

Static polymorphism is binding *subprogram calls* to *subprograms* based on their *statically checked* parameter types. It happens for *overloaded subprograms* which have the same *name* but distinct *parameter profiles*. For each *subprogram call*, the compiler *statically* checks the types of *actual parameters* against the types of *formal parameters* of existing *subprograms* with the same *name* as the *call*, and binds the *call* to the *subprogram* whose *parameter profile* matches. If no such *subprogram* exist, the compiler attempts to bind the *subprogram* which can be matched with the least number of *coersions* of *actual parameters*.

We discussed the simplest form of *dynamic binding of subprogram calls* previously in the *subprograms as parameters* section. *Dynamic polymorphism* is binding *subprogram calls* to *subprograms* based on their *dynamically checked* parameter types. A particular kind of *dynamic polymorphism* is binding a *member function call* based on the *invoking* object type checked at *run time*. In our lectures, we call this particular kind just *polymorphism* for simplicity.

Since *variable parameters* and *type-less parameters* discussed previously do not include *type checking*, they are considered *unsafe* and are usually avoided. Similar techniques exist in other languages provided with *dynamic type checking* which are considered *safer*, such as the `Object` type in Java. *Polymorphism* in *compiled languages* provides a *very safe* and efficient way to simulate *dynamic type binding* where all types are *statically checked*, but *subprogram calls* bind to their *subprograms* based on the *invoking* object type which looks as if it is checked at *run time*.

The following example is a powerful usage of *polymorphism* to achieve *generality*. Suppose we need to create a game where the user (player) can select a hero at the beginning to play with. The game behaviour is basically the same for all heroes, except for some minor details related to each hero type. When a *virtual* function is called through a `Hero*`, the function associated with the *derived* object whose address is hold by the `Hero*` will be executed:

```
class Hero     // Interface
{
public:
    Hero() {}    // The constructor cannot be virtual
    virtual void Fire()=0;
    virtual void Jump()=0;
    virtual void Draw()=0;
    virtual ~Hero() {}    // The destructor should be virtual
};

class SuperMario : public Hero
{
public:
    void Fire() {}
    void Jump() {}
    void Draw() {}    // Draw SuperMario
};
```

```
class SuperMan : public Hero
{
public:
    void Fire() {}
    void Jump() {}
    void Draw() {}    // Draw SuperMan
};

class BatMan : public Hero
{
public:
    void Fire() {}
    void Jump() {}
    void Draw() {}    // Draw BatMan
};

class Game
{
protected:
    Hero* hero;

public:
    Game(Hero* h) {hero=h;}
    void Play() {hero->Draw(); hero->Fire(); hero->Jump();}
};

void TestGame()
{
    Hero* hero = new SuperMan;
    Game game(hero);
    game.Play();
    delete hero;
};
```

Since functions of `class Game` use a `Hero` pointer, it can use objects of its *derived* classes, such as `SuperMan`, and call its functions, without including any declaration of `SuperMan` inside `class Game`. Thus, it is compatible with the SOLID *Liskov substitution principle*:
"Modules that use pointers or references to a base class must be able to use objects of derived classes without knowing them."

Moreover, it is compatible with the SOLID *single responsibility principle* because the definition of `class Game` needs to change only if the behaviour (or logic) of playing the game changes. It does not need to change if the hero changes. The game can support a new hero easily by just deriving it from `class Hero` and passing a pointer to its object to the `Game` constructor.

The `Hero` class acts as a *placeholder*, that is, we never create objects of type `Hero`. Creating objects of type `Hero` is semantically meaningless since it does not represent a particular hero that can be used in the game. We use a `Hero*` only to hold the address of an object from a *derived* class such as `SuperMan`, but we never use it to hold the address of an object of type `Hero`.

To prevent the programmer from creating objects of type `Hero` by mistake, and to increase *readability*, we should provide at least one *pure virtual function* in the class. A *pure virtual function* does not have an implementation and has the characters `=0` after its declaration. A class containing a *pure virtual function* is called *abstract class* and cannot be *instantiated* (we can not create objects of its type). A class which does not contain any *pure virtual function* is called a *concrete class* and can be *instantiated*.

An *abstract class* may contain some data members and some minimal function definitions which are very difficult to change. However, since data and function definitions are usually subject to change, it is better to avoid them completely by making all functions *pure virtual*. An *interface* is an *abstract class* which does not contain any data members, and all its functions are *pure virtual* except for the constructors which are not *virtual* and the destructor which is *virtual* but not *pure*.

Thus, the above example is compatible with the SOLID *dependency inversion principle* because the high level class `Game` depends on *abstraction* of a hero (the `Hero` *interface*), instead of the *concrete* heroes themselves (such as `class SuperMan`).

In Java, all functions are *virtual* by default, so the reserved word `virtual` is not needed. In C#, the reserved word `virtual` is needed and also the reserved word `override` is needed in the declaration of the *overriding* functions of the *derived classes* which implement the `virtual` functions of the *base class*.

Abstraction mainly improves *writability* and *generality*. *Abstraction* is a very general concept. Any defined function `F()` can be considered as an *abstraction* in the sense that the programmer can rewrite its implementation and recompile the new definition without affecting its calls in the remainder of the program. Also, a defined function `F()` can be considered as an *abstraction* in the sense that it can be called several times with different combinations of parameter values.

Any built-in data type such as `int` is considered as an *abstraction* in the sense that different implementations of the same programming language can define it in different ways such that these ways are compatible with the semantic meaning of integer. Its size and implementations of arithmetic operators may change according to the operating system and the machine configuration as well. A user-define data type such as `Student` is considered as an *abstraction* in the sense that the programmer can rewrite its implementation and recompile the new definition without affecting its usage in the remainder of the program. All the above are examples of *static abstraction*.

In this lecture, we are particularly interested in *dynamic abstraction* and we call it just *abstraction* for simplicity. In *dynamic abstraction*, an *abstract* programming construct can refer to different *concrete* constructs at *run time*, such as the *dynamic polymorphism* described above.

5 Simulating polymorphism

The following implementation by Adam Rosenfield simulates *polymorphism*. A *derived class* object aggregates a *base class* object for the purpose of *inheritance*. A *base class* object contains a *virtual table* containing a *derived class* functions addresses which are assigned during creation of a *derived class* object.

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

struct HeroVtable;

struct Hero
{
    HeroVtable* vtable;
};

struct HeroVtable
{
    // The constructor is not virtual so it does not exist here
    void (*Jump)(Hero*);
    void (*Fire)(Hero*, int);
    void (*Display)(Hero*);
    void (*Destroy)(Hero*); // The destructor is virtual
};

void HeroJump(Hero* hero)
{
    hero->vtable->Jump(hero);
}

void HeroFire(Hero* hero, int armor)
{
    hero->vtable->Fire(hero, armor);
}

void HeroDisplay(Hero* hero)
{
    hero->vtable->Display(hero);
}

void HeroDestroy(Hero* hero)
{
    hero->vtable->Destroy(hero);
}
```

```
struct Superman
{
    Hero parent;
    int num_jumps;
    int fired_armor;
};

void SupermanJump(Superman* superman)
{
    superman->num_jumps++;
}

void SupermanFire(Superman* superman, int armor)
{
    superman->fired_armor+=armor;
}

void SupermanDisplay(Superman* superman)
{
    cout<<"Superman jumped "<<superman->num_jumps
        <<" times and fired "<<superman->fired_armor
        <<" shots!"<<endl;
}

void SupermanDestroy(Superman* superman)
{
    // Free resources here
}

HeroVtable superman_vtable=
{
    (void(*)(Hero*)) SupermanJump,
    (void(*)(Hero*,int)) SupermanFire,
    (void(*)(Hero*)) SupermanDisplay,
    (void(*)(Hero*)) SupermanDestroy
};

void SupermanInitialize(Superman* superman)
{
    superman->parent.vtable=&superman_vtable;
    superman->num_jumps=0;
    superman->fired_armor=0;
}
```

```
struct Batman
{
    Hero parent;
    int num_actions;
};

void BatmanJump(Batman* batman)
{
    batman->num_actions++;
}

void BatmanFire(Batman* batman, int armor)
{
    batman->num_actions++;
}

void BatmanDisplay(Batman* batman)
{
    cout<<"Batman performed "<<batman->num_actions
        <<" actions!"<<endl;
}

void BatmanDestroy(Batman* batman)
{
    // Free resources here
}

HeroVtable batman_vtable=
{
    (void(*)(Hero*)) BatmanJump,
    (void(*)(Hero*,int)) BatmanFire,
    (void(*)(Hero*)) BatmanDisplay,
    (void(*)(Hero*)) BatmanDestroy
};

void BatmanInitialize(Batman* batman)
{
    batman->parent.vtable=&batman_vtable;
    batman->num_actions=0;
}
```

The above function pointers casts are valid since all parameters are of *compatible types*. It is also possible to avoid such casts by matching the function protocols of the *derived* and *base* classes and performing a *type cast* of the first parameter at the beginning of each *derived* function.

```
int main()
{
    Superman superman;
    SupermanInitialize(&superman);

    Batman batman;
    BatmanInitialize(&batman);

    Hero* heroes[2];
    heroes[0]=(Hero*)&superman;    // heroes[0]=&superman.parent;
    heroes[1]=(Hero*)&batman;    // heroes[1]=&batman.parent;

    int i;
    for(i=0;i<2;i++)
    {
        HeroFire(heroes[i], 3);
        HeroJump(heroes[i]);
        HeroJump(heroes[i]);
        HeroFire(heroes[i], 4);
    }

    for(i=0;i<2;i++)
    {
        HeroDisplay(heroes[i]);
        HeroDestroy(heroes[i]);
    }

    return 0;
}
```

The above `(Hero*)` *type cast* is possible because `Superman` and `Batman` objects aggregate a `Hero` object as their first field. This *type cast* (*type conversion*) is called an *up cast* because it converts a *derived* object to a *base* object. Converting a *base* object to a *derived* object is called a *down cast*.

The above program outputs the following:

```
Superman jumped 2 times and fired 7 shots!
Batman performed 4 actions!
```

6 Templates

Templates are a powerful C++ mechanism that attempts to achieve *generality* by *code generation*. Code is generated by *text substitution* then compiled. The following program attempts to implement the game we discussed previously using *templates* instead of *polymorphism*. The following program uses *templates* to ask the compiler to generate separate `Game` classes, each class aggregates an object from a *concrete* Hero class:

```
class SuperMario {public: void Fire(){} void Jump(){}};
class SuperMan {public: void Fire(){} void Jump(){}};
class BatMan {public: void Fire(){} void Jump(){}};

template<class ConcreteHero>
class Game
{
private:   ConcreteHero hero;
public:    void Play() {hero.Fire(); hero.Jump();}
};

void TestGame()
{
    Game<SuperMan> g1; g1.Play();
    Game<BatMan> g2; g2.Play();
}
```

The programmer avoids code duplication to increase *writability*, and asks the compiler to duplicate code on-the-fly before compiling. The compiler generates the following code and then compiles it. Note that the compiler did not generate `class GameSuperMario` because it is not referenced:

```
class GameSuperMan
{
private:   SuperMan hero;
public:    void Play() {hero.Fire(); hero.Jump();}
};

class GameBatMan
{
private:   BatMan hero;
public:    void Play() {hero.Fire(); hero.Jump();}
};

void TestGame()
{
    GameSuperMan g1; g1.Play();
    GameBatMan g2; g2.Play();
};
```

As another example on *templates*, consider the following program which sorts an array:

```
template<class T>
void SelectionSort(T* a, int n)
{
   int i,j;
   for(i=0;i<n;i++)
   {
      // Keep the minimum of a[i ... j] in a[i]
      for(j=i+1;j<n;j++)
      {
         if(a[j]<a[i])   // Less than operator of T
         {
            T temp=a[i];   // Copy constructor of T
            a[i]=a[j];   // Assignment operator of T
            a[j]=temp;   // Assignment operator of T
         }
      }
      // Here: a[i] is the minimum of a[i ... n]
      // Hence: a[0... i] are sorted
   }
}
int main()
{
   int a[]={3,5,2,1,7,4,6}; int n=sizeof(a)/sizeof(a[0]);
   SelectionSort(a,n);
   for(int i=0;i<n;i++) cout<<a[i]<<" "; cout<<endl;
   return 0;
}
```

The `SelectionSort()` function is a *template* function which can sort an array whose elements are of type `T` such that type `T` contains the definition of *copy constructor*, *copy assignment* operator, and *less than* operator.

Consider the following program which utilizes the C++ standard template library:

```
#include <vector>
#include <algorithm>  // includes sort()
using namespace std;

int main()
{
   vector<int> v;    // creates a dynamic array object v
   v.push_back(4);   // adds the integer 4 to the end of v
   v.push_back(7);   // adds the integer 7 to the end of v
   v.push_back(2);
```

```
    // creates an iterator object iter which can traverse the elements of v
    // v.begin() is an iterator to the first element
    // v.end() is an iterator to the element after last
    // iterator allows similar processing of different containers (vector, set, ...)
    vector<int>::iterator iter;   // this loop works with any container:
    for(iter=v.begin(); iter!=v.end(); iter++) cout<<*iter;
    cout<<endl;   // prints: 4 7 2

    sort(v.begin(), v.end());   // sorts elements of v (works with any container)
    for(iter=v.begin(); iter!=v.end(); iter++) cout<<*iter;
    cout<<endl;   // prints: 2 4 7
    return 0;
}
```

Suppose we wish to implement similar behaviour in C++ without using the standard libraries, we can proceed as the following:

```
template <class T>
class Vector
{
private:
    int n;
    T a[100];
public:
    Vector() {n=0;}
    void push_back(const T& x) {a[n]=x; n++;}

    typedef T* Iterator;

    Iterator begin() {return &a[0];}
    Iterator end() {return &a[n];}
};

template <class T, class Iterator>
void Sort(Iterator a, Iterator b)
{
    for(Iterator i=a; i!=b; i++)
    {
        for(Iterator j=i+1; j!=b; j++)
        {
            if(*j<*i) {T temp=*i;  *i=*j;  *j=temp;}
        }
    }
}
```

```
int main()
{
   Vector<int> v;
   v.push_back(4); v.push_back(7); v.push_back(2);
   Vector<int>::Iterator iter;
   for(iter=v.begin(); iter!=v.end(); iter++) cout<<*iter;
   cout<<endl;
   Sort<int, Vector<int>::Iterator>(v.begin(), v.end());
   for(iter=v.begin(); iter!=v.end(); iter++) cout<<*iter;
   cout<<endl;
   return 0;
}
```

The only drawback of the previous program is that inside `Sort()`, we are not able to deduce the type `T` from the type `Iterator`. So, we needed to pass the type `T` to `Sort()` as shown in the above code. This issue is handled in the following version:

```
template <class T>
class IteratorBasedOnPointers
{
private:
   T* i;
public:
   typedef T value_type;
   IteratorBasedOnPointers<T> () {}
   IteratorBasedOnPointers<T> (T* const j) {i=j;}
   void operator++(){i++;}
   void operator++(int){i++;}
   operator T*() {return i;}
};

template <class T>
class Vector
{
private:
   int n;
   T a[100];
public:
   Vector() {n=0;}
   void push_back(const T& x) {a[n]=x; n++;}
   typedef IteratorBasedOnPointers<T> Iterator;
   Iterator begin() {return &a[0];}
   Iterator end() {return &a[n];}
};
```

```
template <class Iterator>
void Sort(Iterator a, Iterator b)
{
   for(Iterator i=a; i!=b; i++)
   {
      for(Iterator j=i+1; j!=b; j++)
      {
         if(*j<*i)
         {
            typename Iterator::value_type temp=*i;
            *i=*j; *j=temp;
         }
      }
   }
}
```

The keyword `typename` tells the compiler that `Iterator::value_type` is a type, not a variable. The compiler cannot conclude this fact when it compiles `Sort()` because the type of `Iterator` is not determined in one of the compilations. The compiler compiles a *template* function or class independently of any type, and also it compiles it once for each type substitution.

Suppose we wish to pass the comparison function as argument to the `Sort()` function instead of relying on the definition of the *less than* operator, we can overload the `Sort()` *template* function as follows:

```
template <class Iterator, class LessThan>
void Sort(Iterator a, Iterator b, LessThan IsLess)
{
   for(Iterator i=a; i!=b; i++)
   {
      for(Iterator j=i+1; j!=b; j++)
      {
         if(IsLess(*j, *i))
         {
            typename Iterator::value_type temp=*i;
            *i=*j;
            *j=temp;
         }
      }
   }
}
```

The previous definition of `Sort()` accepts a function which takes two variables of type `T` and returns `true` only if the first one is less than the second one. Additionally, it can accept a *function object* (also called *functor*) which is an object from a class which overloads the parentheses `()` operator to behave similarly to a function:

```
bool LessThan1(int a, int b)
{
   return a<b;
}

class LessThan2
{
public:
   bool operator()(int a, int b)
   {
      return a<b;
   }
};

int main()
{
   // ...
   // Any one of the following lines works
   Sort(v.begin(), v.end());   // Use the less than operator
   Sort(v.begin(), v.end(), LessThan1);   // Use a global function
   Sort(v.begin(), v.end(), LessThan2());   // Use a functor (class object)
   // ...
}
```

Templates are more *efficient* than *polymorphism*, because *templates* do not have the *run time* overheads of implicitly checking the object type and redirecting execution to the intended function.

But, *templates* are not useful when we need to parametrize a template with the *derived* type of a *base class* pointer. Also, *templates* are not useful when defining an array of *base class* pointers that hold the addresses of different *derived classes*. In such cases, *polymorphism* is essential.

Templates are not intended to replace *polymorphism*. *Templates* are more flexible than *polymorphism* in some situations, and *polymorphism* is more flexible than *templates* in other situations. They should be considered as various design tools. Excessive usage of *templates* incorporates *compile time* overheads. Excessive usage of *polymorphism* incorporates *run time* overheads. The designer should make use of suitable combinations of them for a better design.

7 Simulating templates

The basic features of *templates* can be simulated in C and C++ using *preprocessor* directives. The following is an implementation by Andreas Arnold. Suppose we wish to generate the following functions which differ only in one *templated* type:

```
// Computes a:=a+b where a and b are two arrays of length n
void sum_float(float* a, float* b, int n)
{int i; for(i=0;i<n;i++) a[i]+=b[i];}

void sum_int(int* a, int* b, int n)
{int i; for(i=0;i<n;i++) a[i]+=b[i];}

void sum_int64(int64* a, int64* b, int n)
{int i; for(i=0;i<n;i++) a[i]+=b[i];}
```

The first step is to write a macro `TEMPLATE(X,Y)` which concatenates two strings with an underscore in between (to produce function names like above). This macro is implemented in two stages to enable expanding `X` and `Y` first (if they are macros) before concatenating them.

```
// templates.h
#ifndef TEMPLATES_H_
#define TEMPLATES_H_
typedef long long int64; // We wish to avoid types with multiple words
#define CAT(X,Y) X##_##Y
#define TEMPLATE(X,Y) CAT(X,Y)
#endif
```

This file contains the *declaration* of one function assuming that `T` is defined to be a specific type:

```
// sum_as_template.h
#ifdef T
#include "templates.h"
void TEMPLATE(sum,T)(T*, T*, int);
#endif
```

This file contains the *definition* of one function assuming that `T` is defined to be a specific type:

```
// sum_as_template_cpp.h
#ifdef T
#include "templates.h"
// Computes a:=a+b where a and b are two arrays of length n
void TEMPLATE(sum,T) (T* a, T* b, int n)
{
    int i; for(i=0;i<n;i++) a[i]+=b[i];
}
#endif
```

This file contains the *declarations* of all needed functions:

```
// all_possible_sums .h
#ifndef ALL_POSSIBLE_SUMS_H_
#define ALL_POSSIBLE_SUMS_H_
#include "templates.h"
```

```
#ifdef T
#undef T
#endif
#define T float
#include "sum_as_template.h"
```

```
#ifdef T
#undef T
#endif
#define T int
#include "sum_as_template.h"
```

```
#ifdef T
#undef T
#endif
#define T int64
#include "sum_as_template.h"
```

```
#endif
```

This file contains the *definitions* of all needed functions:

```
// all_possible_sums .cpp
#include "templates.h"
#include "all_possible_sums.h"
```

```
#ifdef T
#undef T
#endif
#define T float
#include "sum_as_template_cpp.h"
```

```
#ifdef T
#undef T
#endif
#define T int
#include "sum_as_template_cpp.h"
```

```
#ifdef T
#undef T
#endif
#define T int64
#include "sum_as_template_cpp.h"
```

Only the file all_possible_sums.cpp needs to be compiled, and the file all_possible_sums.h needs to be included in order to use the above *template* functions:

```
// main.cpp

#include <cstdio>
using namespace std;

#include "all_possible_sums.h"

int main(int argc, char** argv)
{
    int ai[3] = {1, 2, 3};
    int bi[3] = {4, 5, 6};

    float af[3] = {1.0, 2.0, 3.0};
    float bf[3] = {1.5, 2.5, 3.5};

    int64 al[3] = {10, 20, 30};
    int64 bl[3] = {40, 50, 60};

    TEMPLATE(sum, int)(ai, bi, 3);      // sum_int(ai, bi, 3);
    TEMPLATE(sum, float)(af, bf, 3);    // sum_float(af, bf, 3);
    TEMPLATE(sum, int64)(al, bl, 3);    // sum_int64(al, bl, 3);

    int i;
    for(i=0;i<3;i++) printf("%d ", ai[i]); printf("\n");
    for(i=0;i<3;i++) printf("%lf ", af[i]); printf("\n");

    return 0;
}
```

8 Dynamic inheritance

Inheritance is a powerful mechanism that allows to extend a *base class*, by creating a *derived class* from it that includes all features of the *base class*, plus possibly more features, which can be viewed as extending the *base class*. Also, it allows reusing the *base class* instead of rewriting its code in the *derived class*. Thus, it is compatible with the SOLID *open/closed principle*:
"Modules should be open for extension but closed for modification."

A class is usually known to be *derived* from another class before *run time*, even in interpreted languages. *Dynamic inheritance* is the ability to dynamically set and change the parent of an object during *run time*. *Dynamic inheritance* is a very powerful mechanism which provides enormous design *flexibility*, but it is directly supported by only few languages which are not commonly used. It can be indirectly simulated in JavaScript using the `prototype` property. Fortunately, there is an effective way to simulate it in C++ using the *decorator design pattern* as follows:

Suppose we need objects of `ClassB` to be *dynamically inherited* from any class *derived* from `ClassA`. `ClassB` should be derived from `ClassA` and also *aggregates* an object of `ClassA*` which will act as its *base class*. Also, it should redefine all functions of `ClassA` and forward such calls to the aggregated `ClassA*` object.

```
class ClassA
{
public:
    virtual void Fun1()=0;
    virtual void Fun2()=0;
    virtual void Fun3()=0;
    virtual ~ClassA() {}
};
```

```
class ClassB : public ClassA
{
private:
    ClassA* parent;

public:
    ClassB(ClassA* p) {parent=p;}
    void SetParent(ClassA* p) {parent=p;}

    void Fun1() {parent->Fun1();}
    void Fun2() {parent->Fun2();}
    void Fun3() {parent->Fun3();}
};
```

`ClassB` is *derived* from `ClassA` only to receive the *virtual* function calls, but actually it acts as if it is *derived* from the *concrete class* of the *aggregated* `ClassA*` object by forwarding to it all `ClassA` function calls.

9 Design and generic programming

Given the declaration of the following *interfaces*:

```
class Button
{
public:
    virtual void Click()=0;
    virtual void DrawNormal()=0;
    virtual void DrawClicked()=0;
    virtual Rectangle GetRect()=0;
    virtual ~Button() {}
};

class Animation
{
public:
    virtual void Animate(Rectangle)=0;
    virtual ~Animation() {}
};
```

`class Button` represents clickable buttons used on graphical user interfaces, while `class Animation` represents an area on screen where animation is played. Both *interfaces* are useful and independently used by other modules.

Suppose that we need to create a button which contains animation. A first attempt is to redefine `class Button` by *inheriting* it from `class Animation` as follows:

```
class Button : public Animation
{
public:
    virtual void Click()=0;
    virtual void DrawNormal()=0;
    virtual void DrawClicked()=0;
    virtual Rectangle GetRect()=0;
    virtual ~Button() {}
};
```

The above approach has a significant drawback. `class Button` now depends upon `class Animation` regardless of whether the button actually has animation. All modules that need non-animated buttons are forced to depend upon animated buttons.

This reduces the *readability* of the programs, introduces additional complexity by incorporating unused *interfaces* in applications which do not need them, and gives more chances to misuse `class Button` if some user calls `Animate()` on a non-animated button.
This violates the SOLID *interface segregation principle*:
"A module should not depend upon an interface which does not use."

A similar issue happens if we attempt to *inherit* `class Animation` from `class Button`. All modules that need animation without button are forced to depend on a useless button *interface*.

Another solution is not to change the original `Button` or `Animation` *interfaces*, and to define a new class as follows:

```
class AnimatedButton : public Button, public Animation
{
public:
    void Animate(Rectangle){} // define animation behaviour here
    void Animate() {Animate(Button::GetRect());}
};
```

This approach has the problem of introducing an *inheritance* dependency between `Animation` and `AnimatedButton` which should not exist. If there are concrete classes `SkyAmination` and `WaterAnimation` derived from `Animation`. There is no way to use these concrete classes in `class AnimatedButton`

A better solution is to use *aggregation* as the following:

```
class AnimatedButton : public Button
{
protected:
    Animation* anim;
public:
    AnimatedButton(Animation* a) {anim=a;}
    void Animate() {anim->Animate(Button::GetRect());}
};
```

We can pass any *concrete class* derived from `Animation` to the constructor of `AnimatedButton`.

Note that `class AnimatedButton` is still an *abstract class* since it *inherits* non-implemented *pure virtual functions* from `class Button`. This solution has the advantage of being close to the *is-a* and *has-a* rule: An animated button *is a* button, so it is *derived* from `class Button`. An animated button *has an* animation, so it *aggregates* an `Animation` object and *delegates* the animation task to it.

Although `class AnimatedButton` is an *abstract class* (not *interface*) because it contains a member variable and an implemented method, it has the same properties of *interfaces* because it does not contain a state (the member variable is actually a stateless *interface*) and the only implemented function *delegates* its task to an *interface*.

Aggregation is usually preferred to *inheritance*, since it gives more flexibility to change the type of *aggregated* objects (as we did in `Game` and `Hero`), keeps each class encapsulated and focused on one task, and keeps class hierarchies simple. This leads to the following principle:
"Prefer object aggregation over class inheritance."

Should `AnimatedButton` aggregate `Button` as well? This has the advantage of using any concrete class derived from `class Button`, such as `WoodButton` and `MetalButton`:

```
class AnimatedButton
{
protected:
    Button* but;
    Animation* anim;
public:
    AnimatedButton(Button* b, Animation* a) {but=b; anim=a;}
    void Click() {but->Click();}
    void DrawNormal() {but->DrawNormal();}
    void DrawClicked() {but->DrawClicked();}
    Rectangle GetRect() {return but->GetRect();};
    void Animate() {anim->Animate(GetRect());}
};
```

However, since `class AnimatedButton` is no more *inherited* from `class Button`, it cannot be used in place of `Button` pointers or references in modules which already use them, which violates the SOLID *Liskov substitution principle*: "Modules that use pointers or references to a base class must be able to use objects of derived classes without knowing them."

Templates overcome these drawbacks as follows:

```
template<class CButton>
class AnimatedButton : public CButton
{
protected:
    Animation* anim;
public:
    AnimatedButton(Animation* a) {anim=a;}
    void Animate() {anim->Animate(CButton::GetRect());}
};
```

This solution enables to use *derived* classes from `Button` by specifying one of them as the *template* parameter, and to use *derived* classes from `Animation` by passing an object from one of them in the constructor, such as:

```
AnimatedButton<WoodButton> obj(new SkyAnimation);
```

Actually, the *template* parameter does not need to be *derived* from `Button`, it just needs to define the functions used inside the *template*. Moreover, objects of `class AnimatedButton` can be used in place of `Button` since they are *derived* from it.

Now, suppose that all we have is a `Button*` and we need to create an `AnimatedButton` object using `SkyAnimation` and the *concrete type* whose address is stored at the given `Button*`. *Templates* cannot help since they require specifying the concrete class name inside the *template* parameter as in the above statement. *Dynamic inheritance* using the *decorator design pattern* offers the solution. Note that it is equivalent to the solution which *aggregates* both objects, except that `class AnimatedButton` is additionally *derived* from `class Button`.

```
class AnimatedButton : public Button
{
protected:
    Button* but;
    Animation* anim;

public:
    AnimatedButton(Button* b, Animation* a) {but=b; anim=a;}
    void Click() {but->Click();}
    void DrawNormal() {but->DrawNormal();}
    void DrawClicked() {but->DrawClicked();}
    Rectangle GetRect() {return but->GetRect();};
    void Animate() {anim->Animate(GetRect());}
};
```

```
Button* AnimatedTheButton(Button* button, Animation* animation)
{
    AnimatedButton obj(button, animation);
    obj->Animate();
    return obj;
}
```

`class AnimatedButton` is *derived* from `class Button` only to receive the *virtual* function calls, but actually it acts as if it is *derived* from the *concrete class* of the *aggregated* `Button* but` object by forwarding to it all `class Button` function calls.