Lecture 5: Abstraction I: Fields & Objects

Introduction

The first object-oriented concept is abstraction, which involves creating a restrictive representation of real-world objects. This is done by defining virtual objects that contain only the necessary attributes and behaviors relevant to a specific purpose. In object-oriented programming (OOP), attributes are represented as fields (or data members), while behaviors are represented as methods (or member functions).

Classes

A class is a blueprint for creating objects. It defines a collection of data and functions that operate on that data. Its general syntax is:

`class` *`identifier`* `{` $\left[\right.$*`access-specifier`* `:` *`members`*$\left.\right]^{+}$ `};`

Access Specifiers Access specifiers determine the visibility of members within and outside the class:

- **public**: Members are accessible everywhere (within the class, through objects, and within *derived classes* [or *subclasses*]).
- **protected**: Members are accessible within the class and derived classes.
- **private**: Members are only accessible within the class.

The access specifier immediately preceding a member provides the member's visibility. If no access specifier is provided, all members are private by default.

Can Access

	Class	Object	Derived Class
`public`	Yes	Yes	Yes
`protected`	Yes	No	Yes
`private`	Yes	No	No

Fields

A field is a non-function member of a class such as a variable, an array, a class, or any other storage structure, and can be of different types.

Instance Fields The default field type is the *instance field*. Each object of a class has its own copy of instance fields. These are declared using normal variable syntax and are initialized in special functions called *constructors*.

Example:

```
class CA
{
  public:  //public accessibility
  //instance fields
  double a;
  double* b;
  double c[20];
};
```

Although an instance field can be constant, it is normally avoided. Likewise, A special instance field is the *this pointer*, which refers to the current instance of the class. It is used when ambiguous naming issues exist between fields and method parameters, and to allow *chain method invocation*, the ability to cascade method callers.

Static Fields A *static field* is a class field, which means it is shared among all objects and accessible without an object. They are declared using normal variable syntax with the *static* keyword preceding them and are initialized outside the class body using the scope resolution operator.

Example:

```
class XA
{
  public:
  //static fields   static int a;
  static int* b;
  static int c[10];
  static const char d;
};

//Initialization of the static fields
int XA::a = 3;
int XA::b = nullptr;
int XA::c[10] = {0,1,2,3,4,5,6,7,8,9};
const char XA::d = 'U';
```

Static constant fields cannot be modified after initialization.

Objects

Once a class is defined, *objects* (variables with the class as its *abstract data type*) can be created. Class members are accessed using the *dot operator* [*member-of operator*] (.) for objects and the *member-of pointer operator* (->) for object pointers.

Example:

```
class SA
{
  public:
  double x;
  double y;
};

int main()
{
  //variable, array, and pointer objects
  SA a, b[20], *c;
  a.x = 5;
  a.y = 3;
  cout << "["<< a.y << ','<< a.x << "]\n";
  cout << "Perimeter:"<< (2 * (a.x + a.y)) << '\n';
  cout << "Area:"<< (a.x * a.y) << '\n';

  for(int i = 0;i < 20;i += 1)
  {
    b[i].y = (i + 2);
    b[i].x = (i + 2);
    cout << "Perimeter:  "<< (2 * b[i].y + 2 * b[i].x) << '\n';
    cout << "Area:  "<< (b[i].y * b[i].x) << "\n\n";
  }
  c = &b[8];
  cout << "[" << c->y << ',' << c->x << "]\n"; //[10,10]
  return 0;
}
```

Objects as Function Parameters & Return Types Objects can be used as return types and parameters of functions. When used as a parameter, it is common to use reference parameters since pass-by-value parameters make copies of their arguments. To copy an object minimally means to copy each of its instance fields, which is an exhaustive process for classes with several fields, meanwhile, depending on the types of fields and the copy procedure, the copy performs some undesirable behavior. Since a reference parameter does not copy, but rather becomes its argument, there is no overhead and unwanted behaviors. Additionally, prohibiting changes to the argument only requires making the parameter constant.

Example:

```
double Perimeter(const SA& a)
{
  return (2 * a.y + 2 * a.x);
}

double Area(const SA* a)
{
  return (a->y * a->x);
}

/*arrays parameters are passed by reference by default*/
void Squares(SA data[],int n,double value)
{
  for(int i = 0;i < n;i += 1)
  {
    data[i].x = value;
    data[i].y = value;
  }
}
```