Chapter 3

User-defined types

3.1 Classes

A class is a **user-defined type**. It is composed of built-in types, other user-defined types and functions. The parts used to define the class are called members. Members are either **data members**, which define the representation of an object of the class, or **function members** (sometimes called methods), which provide operations on such objects. Members can be accessed using the `object.member` notation.

3.1.1 Methods

Methods define provide the operations on objects of a given clas. Some methods have a special meaning:

1. **Constructor**: a method with the same of the class. It is used to initialize (construct) of objects of the class. Constructors can have arguments and you can have more than one constructor for the class.

2. **Destructor**: a method with the same name of the class preceded by the sign `~`.

3. **Operators**: used to provide a conventional notation, similar to the operators defined in built-in types such as `+`, `-`, `*`, `/`, `%`, `[]`, `()`, `&`, `<`, `<=`, `>` and `>=`. This is called *operator overloading.*

It is important to think of class as having an **interface** plus an **implementation**. The interface is the part of the class's declaration that its users access directly. The public part of class is the **user's view of the class**. The implementation details is the **implementer's view**

of the class and usually it is defined in a different file (remember separate compilation in Section 2.3.4). Program 14 shows the interface and program 15 shows the implementation file of class `X`.

Program 14 Class interface

```
class X {
public:
  X(int m);    //constructor
  ~X()  //destructor

  int getM(); // function member
  void addToM(int v);

  X& operator++(); //operator

private:
    int m; // data member
};

X obj(0);           // var is variable of type X
int i = obj.getM();      // retrieve var's data member m
obj.addToM(10);               // call var's member function addToM()
X++;               // call vars operator ++
```

Program 15 Class implementation

```
X::X(int vi):
  m(vi)
  {}

~X::X()
{}

int X::getM()
{
  return m;
}

void X::addToM(int v)
{
  m= m+v;
}

X& X::operator++()
{
  ++this->m;
  return *this;
}
```

The `#define` guard prevent multiple inclusion. The format of the symbol name should be chosen so that it guarantee uniqueness, for example base on the file path in a project's source tree. For example, the file foo/src/bar/baz.h in project foo would have the following guard:

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_

...

#endif  // FOO_BAR_BAZ_H_
```

3.2 Structures

For a class that has only data, the distinction between interface and implementation doesn't make much sense, and there is a simplified version of it, a structure. An structure is a user defined data type which allows you to combine data items of different kinds. Structures are classes where all member data and functions are public. Program 16 shows how to define and use an structure.

3.3 Enumerations

An `enum` is a very simple user-defined type, specifying its set os values as symbolic constants (see Program 17).

3.4 Reference and pointers

You can pass a structure or a class as a function argument in very similar way as you pass any other variable or pointer. You can define pointers to structures or classes in very similar way as you define pointer to any other variable (see Program 18). Remember that to access members through pointer you use the notation `object->member`.

3.5 Classes and free store allocation

Constructors and **Destructors** are conceptually simple but the foundation of most effective C++ programming.

- whatever resource a class object needs to function it acquires in a constructor;

Program 16 Structures

```
struct Date
{
  int  day;
  int  month;
  int  year;

  Date(int d, int m, int y);
  void addDay(int nDays);
};

int main( )
{
  struct Date d1;
  struct Date d2;

  d1.year = 2016;
  d1.month = 03;
  d1.day = 16;

  d2.year = 2016;
  d2.month = 03;
  d2.day = 15;

  cout << "Date 1 day : " << d1.day <<endl;
  cout << "Date 1 month : " << d1.month <<endl;
  cout << "Date 1 year : " << d1.year <<endl;

  cout << "Date 2 day : " << d2.day <<endl;
  cout << "Date 2 month : " << d2.month <<endl;
  cout << "Date 2 year : " << d2.year <<endl;

  Date today(16,03.2016);
  today.addDay(1);

  return 0;
}
```

Program 17 Enumerations

```
enum class Day {monday, tuesday, wednesday, thursday, friday,
    saturday, sunday};
enum class Month {jan=1, feb=2, mar=3, apr=4, may=5; jun=6, jul=7,
    aug=8, set=9, out=10, nov=11, dec=12};
Month m = Month::mar;
```

- during the object's lifetime it may release resources and acquire new ones;
- at the end of the object's lifetime, the destructor releases all resources still owned by the object.

Program 18 User defined types references and pointer

```
void print(const Date& dt);

Date* pdt;

pdt->day = 18;
pdt->month = 1
pdt->ano=2015;
```

Program 19 shows an example using free store memory allocation. If you acquire memory in the constructor you have to delete in the destructor, otherwise you run into memory leak. C++ doesn't provide *automatic garbage collection.* A program that runs "forever" can't afford memory leaks, others will eventually terminate and all the memory used will be returned to the systems. Still, memory leaks are considered a sign of sloppiness and may cause performance problems or even faults if your memory consumption estimate is incorrect. If you do not provide a destructor or a constructor, compiler will provide for you. But it will not acquire or release free store memory.

3.5.1 Copying

The default meaning of copying objects is to all data members from the object being copied to the copy. Consider the class `container` shown in Program 19 what happens if we need a copy of an object of this class? We will write `container c2=c1;`. In this case, `c2.m_elements` will have a copy of `c1.m_elements`, i.e. a copy of the pointer. Two objects sharing a pointer is a source of trouble, for example, whenever `c1` goes out of scope and is destroyed, `c2` pointer will be pointing to an invalid memory or vice-versa.

To solve this problem there are two special methods to control coying: **copy constructors**, called when a new object is initialized from an existing one and **copy assignments** to handle copies. The example can be seen in Program 20.

3.6 Const correctness

C++ allows you to explicitly the keyword `const` to prevent constant objects from getting mutated. For example, if you wanted to create a function `f()`that accepted a `std::string`, plus you want to promise callers not to change the callers `std::string` that gets passed to `f()`, you can have `f()` receive its `std::string` parameter. In the pass by reference-to-const and pass by

Program 19 Classes resource management

```
class container
{
public:
  container(int size):
    m_size(size),
    m_elements(new int[size])
  {
    std::cout << "acquiring memory\n";
    for (int i=0; i<size; ++i) m_elements[i]=0;
  }

  ~container()
  {
    std::cout << "releasing memory\n";
    delete[] m_elements;
  }

  void insert(int elem, int idx)
  { m_elements[idx] = elem; }

  int get(int idx)
  { return m_elements[idx]; }

private:
  int m_size;
  int* m_elements;
};

  void f()
  {
    container c(3);
    c.insert(10,0);
    c.insert(5,1);
    c.insert(1,2);
    std::cout <<c.get(0)<< ", "<< c.get(1)<< ", "<< c.get(2)<< "\n";
  }

  int main()
  {
    f();
    return 0;
  }
```

pointer-to-const cases, any attempts to change the caller's `std::string` within the `f()` functions would be flagged by the compiler as an error at compile-time. In the pass by value case (`f3()`), the called function gets a copy of the caller `std::string` (see Program 21).

Const-ness can also be applied to class methods (see Program 22). Declaring the `const`-ness of a parameter is a form of **type safety**. The benefit of const correctness is that it prevents you

Program 20 Copying

```
class container
{
  ...

  container(container& other):
  m_size(other.m_size),
  m_elements(new int[other.m_size])
  {
    std::cout << "acquiring memory copy constructor\n";
    for (unsigned int i=0; i<m_size; ++i)
      m_elements[i]=other.m_elements[i];
  }

  container& operator=(container& other)
  {
    if (this == &other)
      return *this;

    std::cout << "releasing memory assign operator\n";
    delete [] m_elements;

    std::cout << "acquiring memory assign operator\n";
    m_elements = new int[other.m_size];
    for (unsigned int i=0; i<m_size; ++i)
      m_elements[i]=other.m_elements[i];
    m_size = other.m_size;
    return *this;
  }
};
```

from inadvertently modifying something you didn‘t expect would be modified.

3.7 Class interface

Considering the description of classes and structures, the possibility of separate compilation as well as the public and private options for members and methods, we have to think on how do we design a good interface. Specially the public interface. Of course, it depends on the software design and the problem being solved, but there are a few general principles that we should try to follow:

- keep interfaces complete;
- keep interfaces minimal;
- provide constructors;

Program 21 Const-ness

```
void f1(std::string s)
{ s = "ok"; }

void f2(std::string& sr)
{ sr = "ok"; }

void f3(const std::string& crs)
{ crs = "ooops!"; // error }

void f4(std::string* ps)
{ *ps = "ok"; }

void f5(const std::string* cps)
{
  std::string* pps=new std::string ;
  cps = pps;
  *cps="ooops"; // error
}

int main()
{
  std::string s;
  f1(s);
  f2(s);
  f4(&s);

  const std::string cs="const";
  f1(cs);
  f2(cs);
  f4(&cs);
  f5(&cs);

  return 0;
```

Program 22 Const-ness

```
// mMethod does not change paramenter param1 or the internal
// status of the object
void ClassX::mMethod(const string& param1) const;
```

- use types to provide good argument checking;
- identify non-modifying member functions;
- provide destructors ;
- free all resources in the destructor;
- if you have dynamic allocated members, pay extra attention to constructor, destructor and

verify the need for assign operator and copy constructor.

3.7.1 Classes x structures

Another good question is, how to decide when use a `class` or a `struct`? A very simple rule of thumb is that you should have a real class with an interface and a hidden representation if and only if you can consider an *invariant* for the class. An invariant allows you to say when the object's representation is good and when it isn't. For example: you want to design a type to represent a person by its name and address. If it doens't matter what value you can have in name or address, if you can not think of a "invalid" person, than this is a structure. However, if you decide the semantics should be that first, middle, and last name as parts of the name. You can also decide that the address really has to be a valid address. Either you validate the string, or you break the string up into first address field, second address field, city, state, country, zip code, that kind of stuff, than it should be a class.

3.7.2 Class methods x functions

There is a general rule that helps us decide to choose between methods and external functions to manipulate a class. If the function changes the state of an object, then it ought to be a member of that object. Unfortunately, even this rule says nothing about functions that do not change the state of the object, so we still must decide what to do considering what the function does and also how users might want to call it.

Program 23 shows an example of having the compare operator as function instead of overloading the operator ==.

Program 23 Functions

```
class Student
{
public:
  std::string name() const;
  bool valid() const;
  std::istream& read(std::istream&);
  double grade() const;
private:
  std::string n;
  double midterm, final;
  std::vector<double> homework;
};
bool compare(const Student&, const Student&);
```