Massachusetts Institute of Technology
6.096
Lecture Notes: Lecture 6

User Defined Datatypes

In a variable, one can store a single value of a particular type.
Eg int a;
Allows you to store a single value of type integer in the variable *a*.

In an array, one can store many values but again of the same type.
Eg int a[10];
Allows you to store 10 values of type integer in the array *a*.

In C++, user-defined datatypes enable you to store many values of different datatypes and group them into a single unit. The two main types of these user defined datatypes are structures and classes.

I. **STRUCTURES**

```
struct Employee
{
  int Empno;
  char *Ename;
  char *Add;
};
int main()
{
  Employee Emp1;
  cin>>Emp1.Empno;
  Employee Emp2={10,"John", "77 Mass Ave"};
  cout<<Emp2.Empno<<Emp2.Name<<Emp2.Add;//10John77 Mass Ave
  return 0;
}
```

In the above example, “Employee” is s structure which can store 3 values, Empno, Ename and Add of type integer, string and string respectively.

Each structure type variable (Emp1 and Emp2) in our case, has sub-variables Emp1.Eno, Emp1.Ename, Emp1.Add etc, which can be used like any other variables. Eg we can do `Emp1.Empno++;` to increment the value of Empno by 1 just as we do in regular integers.

Note we cannot do operations on an entire structure variable, except in a few cases (for instance initialization can be done as we did for Emp2)
Eg:
`cin>>Emp1;` or `cout<<Emp1`; are not valid.

Structures are extensions of the ‘struct’ datatype that was used in C. In C++ however, this datatype has been rendered obsolete by classes. Classes combine all of the features of structures, with many features of their own.

II. **Classes**

We can put the above defined structure into a class.

```
class Employee
{
  int Empno;
  char *Ename;
  char *Add;
};
int main()
{
  Employee Emp1;
  cin>>Emp1.Empno;//Not allowed
  Employee Emp2={10,"John", "77 Mass Ave"};//Not Allowed
  cout<<Emp2.Empno<<Emp2.Name<<Emp2.Add;//Not Allowed
  return 0;
}
```

Here Empno, Ename and Add (data inside the class) are known as Data Members of the class. Emp1, and Emp2 (which were earlier structure type variables) are termed as Objects. However, when we try to work with the data inside the objects, we cannot do so. Even basic input output operations do not work. This is because classes have Visibility Modes for the data inside them.

The three main visibility modes of members of a class are-
Public: Accessible by objects of a class.
Private: Not accessible by objects of a class
Protected: Not accessible by objects of a class.

Note:
Members of a class are Private by default
The difference between the Private and Protected visibility modes will come when the topic of inheritance is discussed.

So the above code will work if we add the keyword *public* before the members followed by a colon (:)

```
class Employee
{
 public:
  int Empno;
  char *Ename;
  char *Add;
};
```

```
int main()
{
  Employee Emp1;
  cin>>Emp1.Empno;
  Employee Emp2={10,"John", "77 Mass Ave"};
  cout<<Emp2.Empno<<Emp2.Name<<Emp2.Add;
  return 0;
}
```

However, as will become clear to you in the subsequent lectures, in Object Oriented Programming, we avoid declaring data members in private in order to keep the data "hidden". Thus we should not move the data members to public mode. But, in that mode they are not accessible by objects. How do we then manipulate the data?

Classes have the ability to have functions inside them to manipulate the data. These are known as Member Functions. These can be defined in public mode so that objects can directly access them.

Note:
When we say 'members' of a class, we are referring to both data members and member functions.

E.g

```
class Employee
{
  int Empno;
  char *Ename;
  char *Add;
  public:
  void Input()
  {
    cin>>Empno>>Ename>>Add;
  }
  void Output()
  {
    cout<<Empno<<Ename<<Add;
  }
};
int main()
{
  Employee Emp1;
  Emp1.Input();
  Emp1.Output();
  return 0;
}
```

Member functions can be defined inside as well as outside the class, but it is usually preferable to define them outside the class, especially if they have control structures such as conditional constructs and loops. This is because some compilers tend to give errors when functions containing these are defined within the classes.

To define functions outside the classes, we have the function prototypes within the classes, and the definition outside.

```
class Employee
{
  int Empno;
  char *Ename;
  char *Add;
  public:
  void Input();
  void Output();
};
int main()
{
  Employee Emp1;
  Emp1.Input();
  Emp1.Output();
  return 0;
}
void Employee::Input()
{
  cin>>Empno>>Ename>>Add;
}
void Employee::Output()
{
  cout<<Empno<<Ename<<Add;
}
```

:: is called ‘scope resolution operator’, and is used to indicate that the functions belong to a particular class. For instance, `Employee::Input()` means function Input() belongs to class Employee.
Had we only written `void Input()` and defined it, it would have meant that this function is not related to the class.

Member functions in a class are identical to regular functions- they are defined normally, have a return type, name and arguments. They can also be overloaded like regular functions. Also regardless of the order in which they are defined, every member function in the class can call the other.

Consider the following program which utilizes the same class Employee with a few extensions:

```
class Employee
{
  int Empno;
  char *Ename;
  char *Add;
  float Salary, Comm, TotSal;
  public:
  void Input();
  void Output();
  void ChangeEmpno(int);//function with argument of type int
  void ChangeEmpno2(int =2);//function with default argument
  float Tot_sal();//function with return type float
};
int main()
{
  Employee Emp1;
  Emp1.Input();
  Emp1.Output();
  cout<<Emp1.Tot_Sal();
  Emp1.ChangeEmpno();
  Emp1.ChangeEmpno(458);
  Emp1.ChangeEmpno2(100);
  return 0;
}
void Employee::Input()
{
  cin>>Empno>>Ename>>Add>>Salary>>Comm;
  TotSal=Tot_Sal();//Input() is calling Tot_Sal()
}
void Employee::Output()
{
  cout<<Empno<<Ename<<Add<<Salary<<Comm<<TotSal;
}
void Employee::ChangeEmpno()
{
  int NewEmpno;
  cin>>NewEmpno;
  Empno=NewEmpno;
}
void Employee::ChangeEmpno(int NewEmpno)
{
  Empno=NewEmpno;
}
void Employee::ChangeEmpno2(int IncEmpno)
{
  Empno+=IncEmpno;
}
float Employee::Tot_sal()
{
  return Salary+Comm;
}
```

Passing Class Objects as Arguments to Functions
Similar to other variables, class objects can also be passed as arguments to functions. These can be passed by value or by reference.

Passing by Value:

```
int SampleFunction1 (Employee E)
{
  E.Input();
  E.Output();
}
```

Passing by Reference:

```
int SampleFunction2(Employee &E)
{
  E.Input();
  E.Output();
}
```

Having a class as return-type of a function:
This indicates that we will be returning an object of the class Employee to the calling function when this function is called.

```
Employee SampleFunction3()
{
  Employee E1;
  E1.Input();
  return E1;
}
```

String data type revisited:
The above discussion of classes should make it a bit clearer what was going on with the string data type. In the string header file, a class called string is declared, one of whose data members is an array of characters – just a regular `char[]` (or `char *`). However, the member functions provided by the string class make many string operations much easier.

For instance, with C-style strings (a `char *`), copying a string `str1` to a string `str2` would require allocating `str2`, making sure it has enough space for all the characters of `str1`, and then calling `strcpy(str1, str2)` – not exactly intuitive.

Using the C++ string class, though, we can just create string objects, and then write things like `string str1 = str2;`. The member functions of the string class automatically take care of all the messy memory stuff for us. (This assignment statement makes use of the initialization function of the class – the constructor, which we'll learn about more next time.)

A complete list of member functions can be found at http://www.cppreference.com/wiki/string/start. Also, some special versions of regular operators are defined to work with strings. For instance, addition of strings (`string str2 = str1 + "hi";`) or accessing subscripts of strings (`str1[5]` to get 6th character) are supported. Some good usage examples can be found at http://www.bgsu.edu/departments/compsci/docs/string.html.

Dynamic memory allocation:
The string class can't allocate one static array to hold the characters of a string; it may need to adjust the length of the array if the user appends or removes characters. Thus far, we've only seen how to declare an array of static size – one whose size is known at compile time. To allocate memory for an array whose size is only determined at runtime, we need the `new` operator.

The syntax for the `new` operator is:

```
type_name *ptr = new type_name;
```

`type_name` can be a built-in (primitive) type or a `struct` or `class` type.
For example: `int *newIntPtr = new int;` allocates a new integer, which did not exist in memory before the statement was executed, and returns a pointer to it.

This is not terribly interesting for small data types like integers, but you can also dynamically allocate objects (of user-defined types, i.e. classes) and arrays. If you want to allocate an array of `type_name` values, you can just add the brackets at the end with the number of elements (which can be anything, even a variable, that evaluates to an integer). For example: `char *newStr = new char[25];` allocates memory for a new 25-character string.

Note: `new` always returns a pointer. This form of dynamic memory allocation is one of the primary uses of pointers.

Any memory allocated with `new` must be freed when the data stored there is no longer needed. This is done with the `delete` operator:

```
int *intPtr = new int;
…// Do stuff with intPtr
delete intPtr;    // De-allocates memory for this integer
```

For arrays, use the `delete[]` operator, or else not all the memory of the array will be freed:

```
int *nums = new int[10];
…// Do stuff with nums
delete[] nums;    // De-allocates memory for the entire array
```

Failing to delete memory you allocated causes memory leaks – your program gobbles more and more memory unnecessarily without ever releasing it.