Chapter

15

User-Defined Types

Primitive types are fine for creating simple programs. But, for most tasks, you need more complex **user-defined data types**, such as `string` and `vector`. We can combine multiple data items into a larger unit, called a **structured** type. The types in the standard library, such as `string` and `vector`, are **structured** types.

The C++ **language** has two derived structured data types—a linear **list-type** collection, called an **array**, and a **record-type** collection, called a **structure or class**. The `Date` class shown here is a **structured** user-defined type, consisting of a month, day and year.

Structure Definitions

Programs are often built around **collections of data**, like an **employee record**. Each worker has an employee number, a name, an address, job title and so on. Such types are called **records** (generic CS term) or **structures** (the C/C++ term).

Structures **combine** **related pieces of information** into a composite object which can be manipulated as a unit. C++ has two structured user-defined types: **structures** (or `struct`) and **classes**. In this chapter, we'll begin with the structure-based model.

Here is the C++ **definition** for a `Date` user-defined structure type:

```
#include <string>
struct Date
{
    std::string month;
    int day;
    int year;
};
```

This **defines a new type** which contains **three data members**. (Don't call them fields; the term field has a different meaning in C++). The name (`Date`) is formally known as the **structure tag**. Structure members **do not** all need to be of the same type; we say that structures are **heterogeneous** data types. In `Date`, `month`, is of type `string`, while the others are of type `int`.

Structure definitions are normally found **inside header files**. Thus, all library members (such as the `std::string month`), must be **fully qualified**. It is **illegal** to include the same type (`struct` or `enum`) definition multiple times, even if the definitions are exactly the same. Protect against this by using **header guards** (not shown here).

> *Don't forget the semicolon appearing after the final brace. If you leave it off, you're likely to see a misleading error message pointing to a different area of your code.*

Nested Structures

A structure member may be another structure. This is a **nested structure**. A person has a name and a birthday. We have a `Date`, so we can use it in a `Person` definition.

```
struct Person
{
    std::string name;
    Date birthday;
};
```

Structure Variables

A structure definition **introduces a new type**. Once you have the type definition, you can **define variables**, as you would with any other type.

```
int n;
Date today;
```

These two lines instruct the compiler to **allocate memory** for the `int` variable `n`, and for the `Date` variable `today`. The `Date` variable `today` includes data members that store the values of its `month`, `day` and `year` components. If you were to draw a box diagram of the variable, it would look something like the picture on the left.

Initialization

Just as the `int` variable `n` is **uninitialized**, `day` and `year` in the variable `today` are **also** uninitialized. The `month` member is **default initialized**, because it is a library type.

> *This is the opposite of Java. If we were to create a Date class with a public String field, that field would be uninitialized, while the primitive types would be default initialized.*

Initialize a structure variable with **a list of values**, inside curly braces, separated by commas and ending with a semicolon. This is **aggregate initialization**.

```
Date empty{};
Date mo{"Jan"};
Date bday = {"February", 2, 1950};
```

If you supply no initializers, all members are **default initialized**. In this case, that means that `day` and `year` are set to `0` instead of a random number.

When you use a single initializer, the first data member (`month` in this case) is initialized, while the remaining members are **default initialized**. If you provide extra values, it is a syntax error and **your code will not compile**.

> *In C++11, you don't need =. It is required for C++98.*

Member Initialization

In C++11 (and later) you can provide **in-place initializers**, just like Java:

```
struct Date
{
    std::string month;
    int day = 0;
    int year{0};
};
```

Use legacy ("assignment") initialization (`day`), or uniform initialization (`year`). You **may not use direct initialization** with parentheses instead of braces. Note that `month` does not need an initializer, since it is a library type.

Anonymous Structures

You may also create a **structure variable** along with the definition. This can be useful when you need to group together a pair of variables for immediate use.

```
struct iPair {int a, b;} p1;
struct {int a, b} p2;
```

Here, `p1` is a structure variable, of type `iPair`. When you do this, you may also **omit** the structure tag, as is done for the variable `p2`, creating an **anonymous structure**.

Member Access

Select the members of a structure variable by using the **member access operator**, or, more informally the **dot operator**.

```
cout << bday.month << endl;
```

Here, `bday` is the variable and `month` is the data member. Such **selection expressions are assignable**, so you can modify the components of `bday` like this:

```
Date bday;
bday.month = "February";
bday.day = 2;
bday.year = 1950;
```

Since this is assignment, and **not initialization**, this must appear inside a function.

With a nested structure:

- You can access the nested member in its entirety (aggregate)
- You access the data members of the nested structure, using another level of dots. Here is an example.

```
Date bday{"Febrary", 2};     // No year
Person steve {"Stephen"};    // no birthday
steve.birthday = bday;       // aggregate
steve.birthday.year = 1950; // add year
```

Aggregate Operations

Structure types in C **cannot** automatically perform **all** of the common operations that the built-in types can so we say that such derived types are **second-class types**. C++, however, allows programmers to **define** operations that **work with the structure as a whole**. These are called **aggregate** operations, implemented by **overloaded operators**.

Built-in Aggregate Operations

Four **built-in aggregate operations** work in C and in C++: assignment, initialization, passing parameters and returning structures.

Given a **`Date`** variable:

- You can **assign** it to another variable, just as if it were an **`int`** or **`double`**.
- You can use it to **initialize** another variable.
- You can **pass it to a function** as an argument.
- You can **return it** from a function.

All of these are closely related to assignment.

Unsupported Operations

Here are some things you can and cannot do with "plain" structures:

```
Date d1{"February", 2, 1950}, d2;
d2 = d1;                  // assignment OK
Date d3{d2};              // initialize OK
if (d1 == d2)             // No aggregate comparison
    cout << d1 << endl;   // No aggregate I/O
d1++;                     // No built-in arithmetic
```

- You **cannot compare two structures** using either equality or the relational operators. You must compare the individual data members instead.
- You **cannot automatically display** a structure variable using `cout`; you must access and print the individual data members.
- There is no **built-in arithmetic**.

It is, however, easy to turn each of these operations into an aggregate operation by simply **writing some functions**. We'll look at those shortly.

Structures & Functions

Structures may be **passed** as arguments and **returned from** functions. Simply specify the structure type as parameter or return type. We can use this to get around the inconvenience of the missing structure aggregate operations.

Comparing Structures

Although we **cannot** compare two structures with `==` or `!=`, we can write a function to supply the necessary functionality, like this:

```
bool equals(Date lhs, Date rhs)
{
   return lhs.month == rhs.month &&
        lhs.day == rhs.day  && lhs.year == rhs.year;
}
```

> ***lhs*** *and* ***rhs*** *are shorthand for left-hand-side and right-hand-side.*

The function `equals()` takes two `Date` parameters and returns `true` if they are equal and `false`, otherwise. The common parameter names `lhs` and `rhs` are shorthand for **left-hand-side** and **right-hand**-side. Use the function like this:

```cpp
if (equals(d1, d2)) cout << "equal" << endl;
```

C++ and Pass-by-Value

The two arguments, **d1** and **d2**, are **passed by value**., which means that the parameter variables **lhs** and **rhs** are initialized by **making a copy** of the entire **Date** structure when calling the function.

In this particular case, the **cost** (time and memory) of making that copy is not very high; but, if the structure had more data members, calling this function **could be very expensive**. For structure, class and library types, **avoid that cost** by:

- Use **const reference** if the function **should not** modify the argument.
- Use **non-const reference** if the **intent** is **to modify** the actual argument.

A more correct version of equals would look like this:

```cpp
bool equals(const Date& lhs, const Date& rhs)
{
   return lhs.month == rhs.month &&
        lhs.day == rhs.day  && lhs.year == rhs.year;
}
```

In general, never pass a class or structure type by value to a function.

> *This is a fundamental difference in the way that Java and C++ object types work. In Java and C#, objects have **reference semantics**—the object variables do not contain the actual object members. In C++, objects have **value semantics**; the actual data members are stored inside the object variables.*

Introducing Overloaded Operators

You are certainly familiar with the **equals()** method from Java. However, using it is not quite as convenient as using the built-in operators. Fortunately, in C++, you can redefine the meaning of the built-in operators when they apply to a user-defined type, like **Date**.

Doing so is as easy as replacing the name **equals** with the "magic" incantation here:

```cpp
bool operator==(const Date& lhs, const Date& rhs)
{
   return lhs.month == rhs.month &&
        lhs.day == rhs.day  && lhs.year == rhs.year;
}
```

Nothing else in the function needs to change.

You can do the same thing for all of the built-in relational operators. To make this easier, you can define one operator in terms of another.

Given **operator==** above, here is **operator!=**:

```
bool operator!=(const Date& lhs, const Date& rhs)
{
   return !(lhs == rhs);
}
```

Other Relational Operators

The other relational operators are a little more difficult. Intuitively you can tell whether some **Date** is greater than (appears after) a given **Date**, or, whether some **Date** is less than (appears before) another **Date**.

Writing it in code is a little more difficult, but not impossible. Here's one implementation of an **operator<()** which returns **true** when one **Date** appears before another:

```
bool operator<(const Date& lhs, const Date& rhs)
{
    return ((lhs.year < rhs.year) ||
            (lhs.year == rhs.year &&
             lhs.month < rhs.month) ||
            (lhs.year == rhs.year &&
             lhs.month == rhs.month &&
             lhs.day < rhs.day));
}
```

Output Operators

For homework, you're going to write a **print(Point)** function. That's fine, but really. we want to be able to print **Date**, **Point** and **Triangle** objects the same way we print **int** and **string** values. We do this by defining an output operator like this:

```
ostream& operator<<(ostream& out, const Date& d)
{
    out << d.month << ", " d.day << " " << d.year;
    return out;
}
```

Now, **cout** will work for **Date** objects in the same manner as it does for the built-in types.

The output operator **takes a reference to the output stream** as its first argument, and a **const-ref** to the object you are printing as its second. The operator returns the stream after the object has been inserted into it.

Memorize this pattern because it is the same for printing any user-defined type.

Structured Bindings

C++17 added a new feature to the language that makes it easier to retrieve several returned values from a function. These are called structured bindings. Here is a variation on one of the problems given earlier on a Programming Exam.

> *Write a function* ***quadratic()*** *which computes roots of quadratic equations. A quadratic equation is one of the form:* $ax^2 + bx + c = 0$.
>
> *Your function has three* ***input*** *parameters, the integer coefficients* ***a***, ***b***, *and* ***c***. *The function returns a* ***struct*** *containing two* ***double*** *members:* ***root1*** *and* ***root2***. *Assume that the function has two real roots. The quadratic formula is:*
>
> $$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

You would **write the function** like this (in any version of C++):

```
struct Roots { double root1, root2; };
Roots quadratic(int a, int b, int c)
{
    double determinant = b * b - 4 * a * c;
    return Roots{(-b + sqrt(determinant)) / (2 * a),
                 (-b - sqrt(determinant)) / (2 * a)};
}
```

Prior to C++17 you would **call the function** like this:

```
Roots r = quadratic(1, -3, -4);
cout << "roots->" << r.root1 << ", " << r.root2 << endl;
```

Notice that it is up to the programmer to **"unpack"** the returned structure. With C++17 you can **automatically unpack** the structure into **auto** declared variables like this:

```
auto [r1, r2] = quadratic(1, -3, -4);
cout << "roots->" << r1 << ", " << r2 << endl;
```

Note that the programmer does not need to specify the names or types of the local variables **r1** and **r2**. The structure is unpacked and **root1** is assigned to **r1** and **root2** is assigned to **r2**. They are both automatically declared as type **double**.

Enumerated Types

Single value types are called scalar types. All of the built-in, primitive data types—`int`, `char`, `bool` and `double`—are **scalar** data types. The `Weekday` type shown here is a **user-defined scalar** type which contains only a single simple value.

You may define your own **new scalar types** by listing the **elements in their domain**. Such types are called **enumerated types**.

```
enum class typename { namelist }; // C++ 11 scoped enums
enum typename { namelist };   // traditional plain enums
```

where ***typename*** is the **name** of the type and ***namelist*** is a list of literals representing the values in the domain, separated by commas. The ***namelist*** does not need any semicolons, unlike regular variable definitions. However, you **must end with a semicolon**.

The **scoped enumeration** was added in C++11, while the older type is called an **unscoped or plain enumeration**. In this class we'll use the newer, scoped enumerations.

Defining an Enumerated Type

Here is the `Weekday` type mentioned at the beginning of the lesson. The C++ compiler assigns values to the names. `SUNDAY` is assigned `0`, `MONDAY` is assigned `1`, and so on.

```
enum class Weekday
{
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
};
```

You may also **explicitly specify** the underlying values to any or all of the literals of an enumerated type. The `Coin` type represents U.S. coins where each literal is equal to the monetary value of that coin.

```
enum class Coin
{
    PENNY = 1, NICKEL = 5, DIME = 10,
    QUARTER = 25, HALF_DOLLAR = 50,
    DOLLAR = 100
};
```

If you supply initializers for some values but not others, the compiler will automatically number the remaining literals consecutively after the last.

```
enum class Month
{
    JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG,
    SEP, OCT, NOV, DEC
};
```

Here, **`JAN`** has the value **`1`**, **`FEB`** has the value **`2`**, and so forth up to **`DEC`**, which is **`12`**.

Enumerated Variables

Just like the other types you've seen, you can **create a variable** of an enumerated type and initialize the variable with a scoped member of the type like this:

```
Coin c1 = Coin::PENNY;
// Coin c2 = 1; // error
```

Note that you **can't** initialize the variable **`c`** with its underlying **`int`** representation. The second line in the example above is an error. You may, however, initialize or assign an integral value, by **explicitly** using a **`static_cast`**.

```
Coin c3 = static_cast<Coin>(5); // OK, but why?
Coin c4 = static_cast<Coin>(3); // Just wrong
```

As you can see, this is **error prone**, and turns off the error checking that C++ provides. The variable **`c4`** in the example above is simply **undefined. Don't do this**.

However, if you want to get the "underlying" value of an enumerated type, you can use **`static_cast<int>(c)`** where **`c`** is the **`Coin`** variable shown above. Unlike going the other direction, **this is always safe**.

Enumerated switch Selectors

The **names** of the enumerated values **are not `strings`**; you cannot print them:

```
Month m1{Month::JAN};
cout << m1 << endl; // does not compile
```

Since enumerations are constant integral scalar values, you **can** use **`enum`** variables as **`switch`** selectors. When you match the **`switch`** selector against an enumerated value in the **`case`** label, you must fully qualify the name of the value like **`case Month::JAN`**.

Thus you can write **`to_string()`** like this:

```
string to_string(Month m)
{
    switch (m)
    {
        case Month::JAN: return "January";
        case Month::FEB: return "February";
        . . .
        default: return "INVALID MONTH";
    }
}
```

This function converts a **Month** variable to a **string**.

- Enumerated types are internally just integers: **pass them by value**.
- Each case label must use the **fully qualified enumeration literal**.
- The **default** returns an error if **m** does not match any **Month**. You may want to use an assertion as the linked code does.

Output Operators

You can **directly compare enumerated types using the == and != operators**. You **can** also use the relational operators on enumerated types, but, it is **not very useful**.

With the addition of **to_string()**, you can add an **overloaded output operator** that works on your enumerated type, like this:

```
ostream& operator<<(ostream& out, Month m)
{
    out << to_string(m);
    return out;
}
```

Finish Up

- Complete the **reading exercises (REX)** for this chapter.
- Complete the homework using the **CS50 IDE**. The link is on Canvas.
  a. Make sure you submit the assignment using **make submit**.
  b. Make sure you check the CS150 Homework Console to see that your scores got reported, before the beginning of the next lecture.
- Take the **pre-class reading quiz** on Canvas. You have two attempts.

See you in class or on the Canvas discussion board.