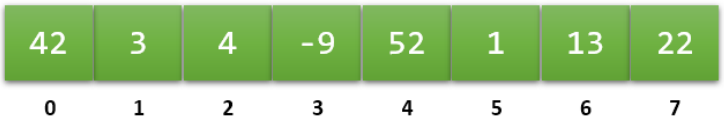
Structured Types

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`. When we combine multiple data items into a larger unit, it is 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. The built-in linear **list-type** collection is called an **array**. In an array, all of the **elements** must be of the same type, so we say that an array is a **homogeneous** structured type. With an array, the elements are accessed by using their **index**, just like the `string` type.



In addition to arrays, programs often **combine related pieces of information** into a composite object which can be manipulated as a unit, such as an **employee record**. Each worker has an employee number, a name, an address, job title and so on. Such types are called **records** (which is a generic Computer Science term) or **structures** (which is the C/C++ specific term).

The **data members** which make up a structure do **not** all need to be of the same type, so we say that a structure is a **heterogenous** data type. The `Date` type shown here is a **structure** type, consisting of a month, day and year. C++ also has more advanced record types, called **classes**, which you'll study later in the semester.

Structure Definitions

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

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

Unlike a variable definition, a structure definition does not create an object in memory. Instead, it **defines or specifies a new type** which contains **three data members**. (You should not call them fields—ala Java— since the term *field* has a slightly different meaning in C++).

The structure name (`Date`) is formally known as the **structure tag**. As mentioned earlier, structure members **do not** all **need** to be of the same type. In `Date`, `month` is of type `string`, while `day` and `year` are of type `int`.

Nested Structures

A structure member may be another type of structure. This is called a **nested structure**. For instance, a person has a name and a birthday. We have a `Date` type, so we can use it as part of a `Person` definition.

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

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 definition multiple times, even if the definitions are exactly the same. Protect against this by using **header guards** (which are 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.*

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;          // uninitialized int variable n
Date today;     // uninitialized Date variable 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 right. 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 Java `Date` class with a public `String` field, that field would be uninitialized, while the primitive types would be default initialized.*

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**.

Initialization

Starting with C++11 you can provide in-definition initializers for each of your data members, just like Java. You should definitely take advantage of this as it will eliminate uninitialized data members.

```
struct Date
{
    std::string month;    // no initializer required
    int day = 0;          // legacy initialization
    int year{0};          // uniform initialization
};
```

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, and it will automatically be initialized by its constructor. However, you **may** explicitly initialize it as well, if you like.

Aggregate Initialization

You may **explicitly initialize** a structure variable by supplying **a list of values**, one for every data member, inside curly braces, separated by commas and ending with a semicolon. This is called **aggregate initialization**.

```
Date birthday = {"February", 2, 1950};
Date empty = {};
```

If you supply no initializers, as with the `Date empty`, then all members are **default initialized**. In this case, that means that `day` and `year` are set to `0` instead of a random number. If the members already have default initializers (from the structure definition), then those default initializers are used instead.

Member Access

You select the individual members of a structure variable by using the member access operator, or, more informally, the **dot operator**, like this:

```
cout << birthday.month << endl;
```

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

```
Date birthday;
birthday.month = "February";
birthday.day = 2;
birthday.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 can access the data members of the nested structure, using another level of "dots". Here is an example.

```
Date groundhog = {"February", 2, 1950};
Person steve;
steve.name = "Stephen";
steve.birthday = groundhog;          // aggregate assignment, or ...
steve.birthday.year = 2023;          // nested dots
```

Aggregate & Unsupported Operations

Structure types in the C programming language 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**. Operations that **work with the structure as a whole** are called **aggregate** operations.

Four **built-in aggregate operations** work in both 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. Here are some things **you can and cannot do** with structure variables:

```
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
```

As you can see:

- 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.

Structure Arguments

Structures may be passed as arguments to functions; they may also be returned from functions. Specify the structure type as the parameter or return type.

We can use this to get around the inconvenience of the missing structure aggregate operations. Although we **cannot** compare two structures with `==` or `!=`, we can write a function to supply the necessary operation, like this:

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

The function `equals()` takes two `Date` arguments and returns `true` if they are equal and `false`, otherwise. Use the function like this:

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

*The parameter names **lhs** and **rhs** are shorthand for left-hand-side and right-hand-side, and are commonly use with functions that mimic the built-in operators.*

By-Value or By-Reference

Let's look at that last example again:

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

Here, 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, we can **avoid that cost** by:

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

Given these guidelines, a **more correct** version of `equals()` would look like this:

```
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.*

A Compare Function

Of course, when comparing dates, we don't just want to know if they are equal. More often we want to know of one date is earlier or later than another.

Instead of writing additional functions, such as `lessThan()` and `greaterThan()`, we can write a single function named `compare()`, which returns `0` if the two dates are equal, and a negative or positive number if the first date is earlier than, or later than the second.

This is a very common, **three-way pattern.** Here's a possible solution

```
bool compare(const Date& lhs, const Date& rhs)
{
   if (lhs.year == rhs.year &&
         lhs.month == rhs.month && lhs.day == rhs.day) {
      return 0;   // all the same
   }
   else if (lhs.year < rhs.year) {
      return -1;  // year is earlier
   }
   else if (lhs.year == rhs.year && lhs.month < rhs.month) {
      return -1;  // year same but month earlier
   }
   else if (lhs.year == rhs.year && lhs.month == rhs.month &&
            lhs.day < rhs.day) {
      return -1;  // year, month same, but day earlier
   } else {
         return 1; // must be after
   }
}
```

Returning Structures

Let's look at a variation on one of the problems given earlier on a Programming Exam. In the original exam, the two roots of the quadratic equation were returned via a pair of **output parameters**. Now that we have structured types, we can write the function a little more naturally, by returning structure.

> *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:*
>
> $$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Using structures, you could **write the function** like this:

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

Then, you could **call the function** like this:

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

Structured Bindings

Let's look at that last function call once again:

```
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. 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**.

With structured bindings, you can **automatically unpack** the structure into a special form of **`auto`** declared variables like this:

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

Note that you do not need to specify the names of the data members in the returned structure variable, nor the 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 `Date` type, which you met in the last lesson, was a user-defined **structured** type.

The `Weekday` type shown here above the `Date` 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**. Enumerated types are based upon another type, in this case the integers. Such types are called **derived** or **compound types**.

The syntax for defining an enumerated type is:

```
enum class type-name { name-list };   // C++ 11 scoped enums
enum type-name { name-list };         // traditional "plain" enums
```

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

The **scoped enumeration** was added in C++11, while the older type is called an **un-scoped or plain enumeration**. In this class we'll use the newer, scoped enumerations, since they will help you avoid all kinds of scope bugs.

Defining Enumerated Types

Here is the definition of 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 integer value used to represent any or all of the literals of an enumerated type. For example, 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, halfDollar = 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`**.

*In previous semesters, the Course Reader used **`UPPER_CASE`** for the enumerated elements, since that is the convention in Java. Looking at several style guides (both Google and the Core Guidelines), we've switched to lower-camelCase in this edition.*

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 `c2` 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 do this?
Coin c4 = static_cast<Coin>(3); // Just wrong. No 3-cent coin
```

As you can see, using `static_cast` in this way 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 a `Coin` variable like those shown above. Unlike casting from `int` to `Coin`, casting from `Coin` to `int` **is always safe**.

Enumerated Output

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. Thus you could write a `to_string()` function like this:

```
string to_string(Month m)
{
    switch (m)
    {
        case Month::jan: return "January";
        case Month::feb: return "February";
        case Month::mar: return "March";
        . . .
        default: return "INVALID MONTH";
    }
}
```

This function converts a `Month` variable to a `string` so you can print it or concatenated it.

- Enumerated types are internally just integers: **pass them by value**.
- Each `case` label must use the **fully qualified enumeration literal**.
- The `default` case returns an error if `m` does not match any `Month`. You may want to use an assertion instead, since it is definitely a programming error if the `default` is ever reached.

Enumerated Input

Enumerated input is more difficult than output, because:

- You have to decide how you want to **encode** the external version.
- You cannot use the input operator (without overloading).
- Because the external representation is text, you can't use a **`switch`**, but have to use sequential **`if`**..**`else if`**..**`else`** statements instead.

```
Month m1;
cin >> m1; // does not compile
```

Here's a version of a **`read_month`** function which reads from input, using the first three letters of the input to convert it to a **`Month`**:

```
istream& read_month(Month& m, istream& in = cin)
{
    string encoded;
    in >> encoded;
    string subs = encoded.substr(0, 3);   // first three characters
    for (char& c : subs) c = to_lower(c); // lowercase

    if (subs == "jan")      m = Month::jan;
    else if (subs == "feb") m = Month::feb;
    else if (subs == "mar") m = Month::mar;
    . . .
    else in.setstate(ios::failbit);   // set failbit
    return in;
}
```

You'd use the function like this:

```
cout << "Enter a month: ";
Month m;
if (read_month(m)) {
    cout << "The month is " << to_string(m) << endl;
}
else {
    cout << "You entered an invalid value." << endl;
}
```