WIKIPEDIA

Composite data type

In computer science, a **composite data type** or **compound data type** is any data type which can be constructed in a program using the programming language's primitive data types and other composite types. It is sometimes called a **structure** or **aggregate data type**,[1] although the latter term may also refer to arrays, lists, etc. The act of constructing a composite type is known as composition. Composite data types are often contrasted with scalar variables.

Contents

C/C++ structures and classes

A `struct` is C's and C++'s notion of a composite type, a datatype that composes a fixed set of labeled **fields** or **members**. It is so called because of the `struct` keyword used in declaring them, which is short for *structure* or, more precisely, *user-defined data structure*.

In C++, the only difference between a `struct` and a class is the default access level, which is *private* for classes and *public* for `struct`s.

Note that while classes and the `class` keyword were completely new in C++, the C programming language already had a crude type of `struct`s. For all intents and purposes, C++ `struct`s form a superset of C `struct`s: virtually all valid C `struct`s are valid C++ `struct`s with the same semantics.

Declaration

A `struct` declaration consists of a list of fields, each of which can have any type. The total storage required for a `struct` object is the sum of the storage requirements of all the fields, plus any internal padding.

For example:

```
struct Account {
   int account_number;
   char *first_name;
   char *last_name;
   float balance;
};
```

defines a type, ***referred*** to as `struct Account`. To create a new variable of this type, we can write `struct Account myAccount;` which has an integer component, accessed by `myAccount.account_number`, and a floating-point component, accessed by `myAccount.balance`, as well as the `first_name` and `last_name` components. The structure `myAccount` contains all four values, and all four fields may be changed independently.

Since writing `struct Account` repeatedly in code becomes cumbersome, it is not unusual to see a `typedef` statement in C code to provide a more convenient synonym for the `struct`.

For example:

```
typedef struct Account_ {
   int    account_number;
   char   *first_name;
   char   *last_name;
   float  balance;
} Account;
```

In C++ code, the `typedef` is not needed because types defined using `struct` are already part of the regular namespace, so the type can be referred to as either `struct Account` or simply `Account`.

As another example, a three-dimensional Vector composite type that uses the floating point data type could be created with:

```
struct Vector {
  float x;
  float y;
  float z;
};
```

A variable named `velocity` with a `Vector` composite type would be declared as `Vector velocity;` Members of the `velocity` would be accessed using a dot notation. For example, `velocity.x = 5;` would set the `x` component of `velocity` equal to 5.

Likewise, a color structure could be created using:

```
struct Color {
  unsigned int red;
  unsigned int green;
  unsigned int blue;
};
```

In 3D graphics, you usually must keep track of both the position and color of each vertex. One way to do this would be to create a `Vertex` composite type, using the previously created `Vector` and `Color` composite types:

```
struct Vertex {
  Vector position;
  Color color;
};
```

Instantiation

Create a variable of type `struct Vertex` using the same format as before: `Vertex v;`

Member access

Assign values to the components of `v` like so:

```
v.position.x = 0.0;
v.position.y = 1.5;
v.position.z = 0.0;
v.color.red = 128;
v.color.green = 0;
v.color.blue = 255;
```

Primitive subtype

The primary use of `struct` is for the construction of complex datatypes, but sometimes it is used to create primitive structural subtyping. For example, since Standard C requires that if two structs have the same initial fields, those fields will be represented in the same way, the code

```
struct ifoo_old_stub {
   long x, y;
};
struct ifoo_version_42 {
   long x, y, z;
   char *name;
   long a, b, c;
};
void operate_on_ifoo(struct ifoo_old_stub *);
struct ifoo_version_42 s;
. . .
operate_on_ifoo(&s);
```

will work correctly.

Type signature

Type signatures (or Function types) are constructed from primitive and composite types, and can serve as types themselves when constructing composite types:

```
typedef struct {
    int x;
    int y;
} Point;

typedef double (*Metric) (Point p1, Point p2);

typedef struct {
    Point centre;
    double radius;
    Metric metric;
} Circle;
```

See also

- Object composition

- struct (C programming language)
- Scalar (mathematics)

References

1. Howe, Denis. "The Free On-line Dictionary of Computing" (http://dictionary.reference.com/browse/aggregate%20type). *Dictionary.com*. Retrieved 1 February 2016.