Unions

Some languages — C and C++, in particular —provide a data structure called a **union** that is similar to a structure but differs from it in that the members of a structure are allocated different memory locations, whereas the members of a union share memory; that is, they all have the same memory address. In C++, the syntax of a union is the same as that for a struct except that the word `struct` is replaced by `union`:

```
union TypeName      // TypeName is optional
{
   declarations of members
};
```

Unions can be used to build structures that have **fixed parts** and **variant parts**. All objects of such a type have the same members in corresponding fixed parts but there may be different members in the variant parts. Such structures are called **variant records** in some languages such as Pascal and are useful in certain specialized problems. To illustrate, suppose computer-usage records for employees at a computer center have one of three structures:

Support staff
Id number
Password
Resource limit
Resources used to date
Department code

Research
Id number
Password
Account number
Security clearance (1–10)
Department code

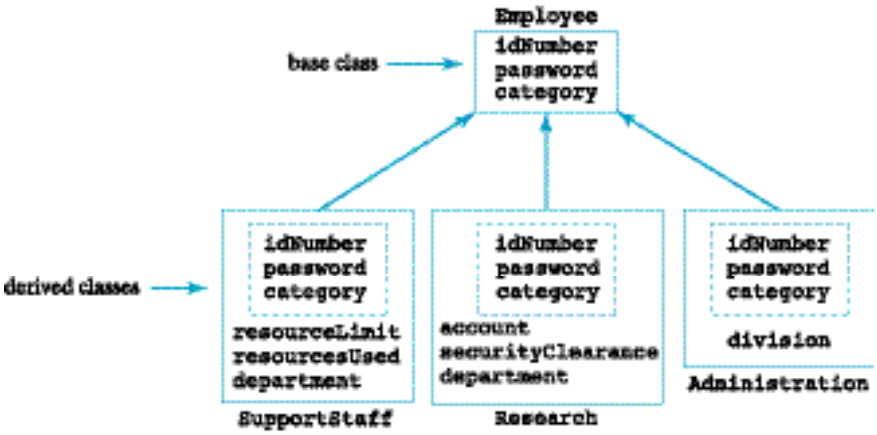
Administration
Id number
Password
Division

All of these structures can be incorporated into a single struct by storing the common information (id number and password) in the fixed part of the struct and the other information in a union whose members are structs containing the varying parts in these structures:

```
struct SupportInfo
{                          // support-person's
  double resourceLimit,    //  limit on computer resources
         resourcesUsed;    //  resources used to date
  char department;         //  department in which employed
};
struct ResearchInfo
{                          // researcher's
  int account,             //  research account
  securityClearance;       //  security clearance: 1-10
  char department;         //  department in which employed
};
struct AdministrationInfo
{                          // administrator's
  char division;           //  division
};
struct ComputerUsageInfo
{                          // user's
  unsigned idNumber;       //  id number
  string password;         //  password
  char category;           //  tag: 'S' = support,
                           //       'R' = research,
                           //       'A' = administration
  union                    //  varying info
  {
    SupportInfo support;
    ResearchInfo research;
    AdministrationInfo administration;
  };
};
```

Note that a member `category` has also been added as a *tag field* to specify which of the three variants is in effect for a given employee.

In OOP languages, structures like this are not needed because *inheritance* makes it possible to encapsulate the fixed information in a *base class* and from it *derive a class* for each variant. As we show in Chap. 10, the OOP approach would be to define a base class containing the common information (id number, password, and category), and then derive classes that inherit these members (and operations) from the base class and contain new members (and operations) for the attributes peculiar to that class. We might picture this as follows:



Structs in Memory

We have seen that to store an array, sufficient memory is allocated to store all the array elements, and that each array reference involves two steps: First, an address translation must be performed to locate where that array element is stored, and second, the bit string stored there must be interpreted in the manner prescribed by the type specification for the array elements. Similarly, storing a structure also requires sufficient memory to store all of the members that comprise the structure. Address translation is again required to determine the location in which a particular member is stored, but this address translation is slightly more complex than for arrays because different members usually require a different number of bytes for storage. Also, unlike arrays, different interpretations of the bit strings are usually required for different members, because the members of a structure need not be of the same type.

To illustrate, consider customer records that consist of a customer's account number, name, an account balance, and the type of account,

```
enum AccountType {WHOLESALE, EDUCATIONAL,
                  REGULAR, HIGH_RISK};
struct Customer
{
  int number;
  char name[20];
  double balance;
  AccountType account;
};
```

```
Customer c;
```

and suppose that `int`s and `enum`s are stored in 4 bytes, `char`s in 1 byte, and `double`s in 8 bytes. This definition of struct `c` instructs the compiler to reserve a block of 40 bytes to store such a struct, the first of which, as for arrays, is called the *base address*. The four bytes beginning at this base address might then be allocated to `c.number`, the next 20 bytes to `c.name`, the next 8 bytes to `c.balance`, and the next 4 bytes to `c.account` as pictured on the next page. When one of these members is accessed, the bit string stored in the associated bytes is interpreted according to the type specified for that field in the struct declaration.

EXERCISES

For Exercises 1–5, assume that values of type `char` are stored in one byte, `int`s in 4 bytes, `double` values in 8 bytes. Give diagrams like those in the text showing where each field of the following structs would be stored:

1.
```
struct Date
{
   char month[8];
   int day, year;
};
```

2.
```
struct Point
{
   double x, y;
};
```

3,
```
struct ClassRecord
{
   int snumb;
   char name[16];
   char sex;
   int testScore[5];
};
```

4.
```
struct StudentRecord
{
   int snumb; char[16] name;
   GradeInfo grades;
   double finalNumScore;
   char letterGrade;
};
```

where the type `GradeInfo` is declared by

```
struct GradeInfo
{
   double homework, tests, exam;
};
```

5,
```
struct Transaction
{
   char customerName[24];
   int customerNumber;
   NumericDate transDate;
   char transType;// tag
   union
   {
      double deposit;        // tag = 'D'
      double withdrawal;     // tag = 'W'
      LoanInfo loan;         // tag = 'L'
      TransferInfo transfer; // tag = 'T'
   };
};
```

where types `NumericDate`, `LoanInfo`, and `TransferInfo` are declared by:

```
struct NumericDate
{
   int month, day, year;
};

struct LoanInfo
{
   int number;
   double payment, interest, newbalance;
};

struct TransferInfo
{
   int account;
   double amount;
   char code;
};
```