16

16 Enum, Struct, and Union

This chapter introduces three simple kinds of programmer defined data types.

You aren't limited to the compiler provided char, int, double data types and their derivatives like arrays. You can define your own data types. Most of Part IV of this text is devoted to various forms of programmer defined type. Here we introduce just three relatively simple kinds. These are almost the same as the equivalents provided in C; in contrast, C has no equivalent to the more sophisticated forms of programmer defined type introduced in Part IV.

Enumerated types, treated in section 16.1, are things that you should find useful in that they can improve the readability of your programs and they allow the compiler to do a little extra type checking that can eliminate certain forms of error. Although useful, enumerated types are not that major a factor in C++ programming.

Structs are the most important of the three language elements introduced here, section 16.2. They do have wider roles, but here we are interested in their primary role which is the grouping of related data.

Unions: treat them as a "read only" feature of C++. You will sometimes see unions being employed in library code that you use, but it is unlikely that you will find any real application for unions in the programs that you will be writing.

16.1 ENUMERATED TYPES

You often need to use simple integer constants to represent domain specific data. For example, suppose you needed to represent the colour of an automobile. You could have the following:

```
const int  cRED   = 0;
const int  cBLUE = 1;
…

int auto_colour;
```

```
auto_colour = cBLUE;
```

This is quite workable. But there are no checks. Since variable `auto_colour` is an integer, the following assignments are valid:

```
auto_colour = -1;
…
auto_colour = rand();
```

But of course, these statements lose the semantics; variable `auto_colour` doesn't any longer represent a colour, it is just an integer.

It is nice to be able to tell the compiler:

> *"Variable auto_colour is supposed to represent a colour, that is one of the defined set of choices RED, BLUE, …."*

and then have the compiler check, pretty thoroughly, that `auto_colour` is only used in this way throughout the code.

enums and type checking

This is the role of `enum`s or enumerated types. If you think how they are actually implemented, they are just an alternative way of declaring a set of integer constants and defining some integer variables. But they also introduce new distinct types and allow the compiler to do type checking. It is this additional type checking that makes enums worthwhile.

The following is a simple example of an enum declaration:

```
enum Colour { eRED, eBLUE, eYELLOW, eGREEN, eSILVERGREY,
                    eBURGUNDY };
```

Naming convention for enums

The entries in the enumeration list are just names (of constant values). The same rules apply as for any other C++ names: start with a letter, contain letters digits and underscores. However, by convention, the entries in the enum list should have names that start with 'e' and continue with a sequence of capital letters. This makes enum values stand out in your code.

With `Colour` now defined, we can have variables of type `Colour`:

```
Colour auto_colour;
…
auto_colour = eBURGUNDY;
```

The compiler will now reject things like `auto_colour = 4`. Depending on the compiler you are using you may get just "Warning – anachronism", or you may get an error (really, you should get an error).

enumerators

What about the "enumerators" `eRED`, `eBLUE` etc? What are they?

Really, they are integers of some form. The compiler may chose to represent them using shorts, or as unsigned chars. Really, it isn't any of your business how your compiler represents them.

The compiler chooses distinct values for each member of an enumeration. Normally, the first member has value 0, the second is 1, and so forth. So in this example, `eRED` would be a kind of integer constant 0, `eSILVERGREY` would be 4.

Note the effect of the type checking:

```
auto_colour = 4;  // Wrong, rejected or at least
                  //        warned by compiler
auto_colour = eSILVERGREY; // Fine, set auto_colour to
                  //        (probably) the value 4
```

It isn't the values that matter, it is the types. The value 4 is an integer and can't be directly assigned to a `Colour` variable. The constant `eSILVERGREY` is a Colour enumerator and can be assigned to a `Colour` variable.

You can select for yourself the integer values for the different members of the enumeration, provided of course that you keep them all distinct. You won't have cause to do this yourself; but you should be able to read code like:

```
enum MagicEnum { eMERLIN = 17, eGANDALF = 103, eFRED = 202 };
```

Treat this as one of those "read only" features of C++. It is only in rare circumstances that you want to define specific values for the members of the enumeration. (Defining specific values means that you aren't really using the enum consistently; at some places in your code you intend to treat enums as characters or integers.)

Output of enums

Enums are fine in the program, but how do you get them transferred using input and output statements?

Well, with some difficulty!

An enum is a form of integer. You can try:

```
cout << auto_colour;
```

and you might get a value like 0, or 3 printed. More likely, the compiler will give you an error message (probably something rather obscure like "ambiguous reference to overloaded operator function"). While the specific message may not be clear, the compiler's unhappiness is obvious. It doesn't really know how you want the `enum` printed.

You can tell the compiler that it is OK to print the enum as an integer:

```
cout << int(auto_colour);
```

The function like form `int(auto_colour)` tells the compiler to convert the data value `auto_colour` to an integer. The statement is then `cout << integer` which the compiler knows to convert into a call to a `PrintInteger()` routine. (The compiler doesn't need to generate any code to do the conversion from `enum` to integer. This conversion request is simply a way for the programmer to tell the compiler that here it is intended that the `enum` be regarded as just another integer).

Printing an `enum` as an integer is acceptable if the output is to a file that is going to be read back later by the program. Human users aren't going to be pleased to get output like:

```
Engine:    1.8 litre
Doors:     4
Colour:    5
```

If you are generating output that is to be read by a human user, you should convert the enum value into an appropriate character string. The easiest way is to use a switch statement:

```
    switch(auto_colour) {
eRED:       cout << "Red";  break;
eBLUE:      cout << "Blue"; break;
            …
eBURGUNDY:
            cout << "Burgundy"; break;
            }
```

Input

If your program is reading a file, then this will contain integer values for enums; for example, the file could have the partial contents

```
1.8 4      5
```

for an entry describing a four door, burgundy coloured car with 1.8 litre engine But you can't simply have code like:

```
double engine_size;
int num_doors;
Colour auto_colour;
…
input >> engine_size;
input >> num_doors ;
input >> auto_colour;
```

The compiler gets stuck when it reaches `input >> auto_colour;`. The compiler's translation tables let it recognize `input >> engine_size` as a form of "input gives to double" (so it puts in a call to a `ReadDouble()` routine), and similarly `input >> num_doors` can be converted to a call to `ReadInt()`. But the compiler's standard translation tables say nothing about `input >> Colour`.

The file contains an integer; so you had better read an integer:

```
int temp;
input >> temp;
```

Now you have an integer value that you hope represents one of the members of the `enum Colour`. Of course you must still set the `Colour` variable `auto_colour`. You can't simply have an assignment:

```
auto_colour = temp;
```

because the reason we started all of this fuss was to make such assignments illegal!

Here, you have to tell the compiler to suspend its type checking mechanisms and trust you. You can say "*I know this integer value will be a valid Colour, do the assignment.*" The code is

```
auto_colour = Colour(temp);
```

Input from file will use integers, but what of input from a human user?

Normally, if you are working with enumerated types like this, you will be prompting the user to make a selection from a list of choices:

```
Colour GetColour()
{
    cout << "Please enter preferred colour, select from "
           << endl;
    cout << "1\tRed" << endl;
    cout << "2\tBlue" << endl;
    …
    cout << "6\tBurgundy" << endl;

    for(;;) {
           int choice;
           cin >> choice;
           switch(choice) {
case 1:    return eRED;
case 2:    return eBLUE;
           …
case 6:    return eBURGUNDY;
           }
           cout << "There is no choice #" << choice << endl;
```

```
            cout << "Please select from 1 Red, 2 Blue … "
                         " 6, Burgundy" << endl;
    }
}
```

(Note, internally `eRED` may be 0, `eBLUE` may be 1 etc. But you will find users generally prefer option lists starting with 1 rather than 0. So list the choices starting from 1 and make any adjustments necessary when converting to internal form.)

Other uses of enums

You do get cases like `Colour auto_colour` where it is appropriate to use an enumerated type; but they aren't that common (except in text books). But there is another place where enums are very widely used.

Very often you need to call a routine specifying a processing option from a fixed set:

DrawString – this function needs to be given the string to be displayed and a style which should be one of
- Plain
- Bold
- Italic
- Outline

AlignObjects – this function needs to be given an array with the object identifiers, a count, and an alignment specification which should be one of
- LeftAligned
- Centred
- RightAligned

You can define integer constants:

```
const int cPLAIN = 0;
const int cBOLD = 1;
…
```

and have an integer argument in your argument list:

```
void DrawString(char txt[], int style);
```

but the compiler can't check that you only use valid styles from the set of defined constants and so erroneous code like

```
DrawString("silly", 12345);
```

gets through the compiler to cause problems at run time.

But, if you code using an enumerated type, you do get compile time checks:

```
enum TextStyles { ePLAIN, eBOLD, eITALIC, eOUTLINE };

void DrawString(char txt[], TextStyles style);
```

Now, calls like:

```
DrawString("Home run", eBOLD);
```

are fine, while erroneous calls like

```
DrawString("Say what?", 59);
```

are stomped on by the compiler (again, you may simply get a warning, but it is more likely that you will get an error message about a missing function).

16.2 STRUCTS

The need for "structs"

It is rare for programs to work with simple data values, or even arrays of data values, that are individually meaningful. Normally, you get groups of data values that belong together.

Let's pick on those children again. This time suppose we want records of children's heights in cm, weights in kilos, age (years and months), and gender. We expect to have a collection of about one thousand children and need to do things like identify those with extreme heights or extreme weights.

We can simply use arrays:

```
const int   kMAXCHILDREN = 1000;

double      heights[kMAXCHILDREN];
double      weights[kMAXCHILDREN];
int         years[kMAXCHILDREN];
int         months[kMAXCHILDREN];
char        gender[kMAXCHILDREN];
…
// read file with data, file terminated by sentinel data
// value with zero height
count = 0;
infile >> h;
while(h > 0.0) {
    heights[count] = h;
    infile >> weights[count] >> years[count] >>
           months[count] >> gender[count];
```

```
        count++;
        infile >> h;
        }
…
```

Now `heights[5]`, `weights[5]`, …, `gender[5]` all relate to the same child. But if we are using arrays, this relationship is at most implicit.

There is no guarantee that we can arrange that the data values that logically belong together actually stay together. After all, there is nothing to stop one from sorting the heights array so that these values are in ascending order, at the same time losing the relationship between the data value in `heights[5]` and those in `weights[5]` … `gender[5]`.

Have to be able to group related data elements

A programming language must provide a way for the programmer to identify groups of related data. In C++, you can use `struct`.

Declaring structs

A C++ struct declaration allows you to specify a grouping of data variables:

```
struct Child {
    double height;
    double weight;
    int    years;
    int    months;
    char   gender;
};
```

After reading such a declaration, the compiler "knows what a `Child` is"; for the purposes of the rest of the program, the compiler knows that a `Child` is a data structure containing two doubles, two integers, and a character. The compiler works out the basic size of such a structure (it would probably be 25 bytes); it may round this size up to some larger size that it finds more convenient (e.g. 26 bytes or 28 bytes). It adds the name `Child` to its list of type names.

This grouping of data is the primary role of a struct. In C++, structs are simply a special case of classes and they can have perform roles other than this simple grouping of data elements. However it is useful to make a distinction. In the examples in this book, structs will only be used as a way of grouping related data elements.

"record", "field", "data member"

The term "record" is quite often used instead of struct when describing programs. The individual data elements within a struct are said to be "fields", or "data members". The preferred C++ terminology is "data members".

Defining variables of struct types

A declaration doesn't create any variables, it just lets the compiler know about a new type of data element that it should add to the standard char, long, double etc. But, once a struct declaration has been read, you can start to define variables of the new type:

Definitions of some variables of struct type

```
Child      cute;
Child      kid;
Child      brat;
```

and arrays of variables of this type:

```
Child       surveyset[kMAXCHILDREN];
```

Each of these `Child` variables has its own doubles recording height and weight, its own ints for age details, and a char gender flag.

Initialization of structs

The definition of a variable of struct type can include the data needed to initialize its data members:

```
Child example = {
    125.0, 32.4, 13, 2, 'f'
};
```

An instance of a struct can be defined along with the declaration:

```
struct Rectangle {
    int left, top;
    int width, height;
} r1, r2, r3;
```

This declares the form of a `Rectangle` and defines three instances. This style is widely use in C programs, but it is one that you should avoid. A declaration should introduce a new data type; you should make this step separate from any variable definitions. The construct is actually a source of some compile-time errors. If you forget the ';' that terminates a structure declaration, the compiler can get quite lost trying to interpret the next few program elements as being names of variables (e.g. the input `struct Rect { … } struct Point {…} int main() { … }` will make a compiler quite unhappy).

Accessing data members of a variable of a struct type
Fields of a variable identified using compound names

Programs have to be able to manipulate the values in the data members of a struct variable. Consequently, languages must provide a mechanism for referring to a particular data member of a given struct variable.

Most programming languages use the same approach. They use compound names made up of the variable name and a qualifying data member name. For example, if you wanted to check whether brat's height exceeded a limit, then in C++ you would write:

```
if(brat.height > h_limit)
    …
```

Similarly, if you want to set cute's weight to 35.4 kilos, you would write:

```
cute.weight = 35.4;
```

Assignment of structs

Most statements and expressions will reference individual data members of a struct, but assignment of complete structures is permitted:

```
Child Tallest;
```

```
Tallest.height = 0;
for(int i = 0; i < NumChildren; i++)
    if(surveyset[i].height > Tallest.height)
            Tallest = surveyset[i];
```

Compilers generally handle such assignments by generating code using a "blockmove". A blockmove (which is often an actual instruction built into the machine hardware) copies a block of bytes from one location to another. The compiler knows the size of the structs so it codes a blockmove for the appropriate number of bytes.

Struct and enum declarations in C libraries

In C++, a struct declaration introduces a new type. Once you have declared:

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

You can define variables of type `Point`:

```
Point p1, p2;
```

In C, `struct` (and `enum`) declarations don't make the `struct` name (or `enum` name) a new type. You must explicitly tell the compiler that you want a new type name to be available. This is done using a `typedef`. There are a variety of styles. Two common styles are:

typedef

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

typedef struct Point Point;
```

or:

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

Similarly, enums require typedefs.

```
enum Colour { eRED, eBLUE, eGREEN };

typedef enum Colour Colour;
```

You will see such typedefs in many of the C libraries that you get to use from your C++ programs.

Structs and functions

A function can have arguments of struct types. Like simple variables, structs can be passed by value or by reference. If a struct is passed by value, it is handled like an assignment – a blockmove is done to copy the bytes of the structure onto the stack. Generally, because of the cost of the copying and the need to use up stack space, you should avoid passing large structs by value. If a function uses a struct as an "input parameter", its prototype should specify the struct as const reference, e.g.:

```
void       PrintChildRecord(const struct& theChild)
{
    cout << "Height " << theChild.height << …
    …
}
```

Functions can have structs as their return values. The following illustrates a function that gets an "car record" filled in:

```
struct car {
    double engine_size;
    int    num_doors;
    Colour auto_colour;
};

car GetAutoDetails()                                          // Function with a struct as a returned value
{
    car    temp;                                              // Local variable of return type defined
    cout << "Select model, GL (1), GLX (2), SX (3), TX2(4) : "
;
    int model;
    cin >> model;
    while((model < 1) || (model >4)) {
           cout << "Model value must be in range 1…4" << endl;
           cout << "Select model :"
           cin >> model;
           }
    switch(model) {                                           // Data entered into fields of local struct variable
case 1:    temp.engine_size = 1.8; temp.num_doors = 3; break;
case 2: …
    }
    temp.colour = GetColour();

    return temp;                                              // return the local struct as the value
}
```

Although this is permitted, it should not be overused. Returning a struct result doesn't matter much with a small structure like `struct car`, but if your structures are large this style becomes expensive both in terms of space and data copying operations.

Code using the `GetAutoDetails()` function would have to be something like:

```
car purchasers_choice;
```

```
…
purchasers_choice = GetAutoDetails();
```

The instructions generated would normally be relatively clumsy. The stack frame setup for the function would have a "return value area" sufficient in size to hold a `car` record; there would be a separate area for the variable `temp` defined in the function. The return statement would copy the contents of `temp` into the "return value area" of the stack frame. The data would then again be copied in the assignment to `purchasers_choice`.

If you needed such a routine, you might be better to have a function that took a reference argument:

```
void GetAutoDetails(struct car& temp)
{
    cout << "Select model, GL (1), GLX (2), SX (3), TX2(4) : "
;
    …
    temp.colour = GetColour();
    return ;
}
```

with calling code like:

```
car purchasers_choice;
…
GetAutoDetails(purchasers_choice);
```

16.3 UNIONS

Essentially, unions define a set of different interpretations that can be placed on the data content area of a struct. For you, "unions" should be a "read-only" feature of C++. It may be years before you get to write code where it might be appropriate for you to define a new union. However, you will be using libraries of C code, and some C++ libraries, where unions are employed and so you need to be able to read and understand code that utilizes unions.

Unions are most easily understood from real examples The following examples are based on code from Xlib. This is a C library for computers running Unix (or variations like Mach or Linux). The Xlib library provides the code needed for a program running on a computer to communicate with an X-terminal. X-terminals are commonly used when you want a multi-window style of user interface to Unix.

An X-terminal is a graphics display device that incorporates a simple microprocessor and memory. The microprocessor in the X-terminal does part of the work of organizing the display, so reducing the computational load on the main computer.

When the user does something like move the mouse, type a character, or click an action button, the microprocessor packages this information and sends it in a message to the controlling program running on the main computer.

In order to keep things relatively simple, all such messages consist of a 96 byte block of data. Naturally, different actions require different data to be sent. A mouse movement needs a report of where the mouse is now located, a keystroke action needs to be reported in terms of the symbol entered.

Xlib-based programs use `XEvent` unions to represent these 96 byte blocks of data. The declaration for this union is

union declaration

```
typedef union _XEvent {
   int  type;
   XAnyEvent   xany;
   XButtonEvent       xbutton;
   XMotionEvent       xmotion;
   XCreateWindowEvent xcreatewindow;
   …
   …
} XEvent;
```

This declaration means that an `XEvent` may simply contain an integer (and 92 bytes of unspecified data), or it may contain an `XAnyEvent`, or it may contain an `XButtonEvent`, or …. There are about thirty different messages that an Xterminal can send, so there are thirty different alternative interpretations specified in the union declaration.

Each of these different messages has a struct declaration that specifies the data that that kind of message will contain. Two of these structs are:

Declaration of alternative structs that can be found in an XEvent

```
   typedef struct {
       int type;
       unsigned long serial;
       Bool send_event;
       Display *display;
       Window window;
       Window root;
       Window subwindow;
       Time time;
       int x, y;
       int x_root, y_root;
       unsigned int state;
       unsigned int button;
       Bool same_screen;
   } XButtonEvent;

   typedef struct {
       int type;
       unsigned long serial;
       Bool send_event;
```

```
    Display *display;
    Window window;
    int x, y;
    int width, height;
    int border_width;
    Bool override_redirect;
} XCreateWindowEvent;
```

As illustrated in Figure 16.1, the first part of any message is a type code. The way that the rest of the message bytes are used depends on the kind of message.

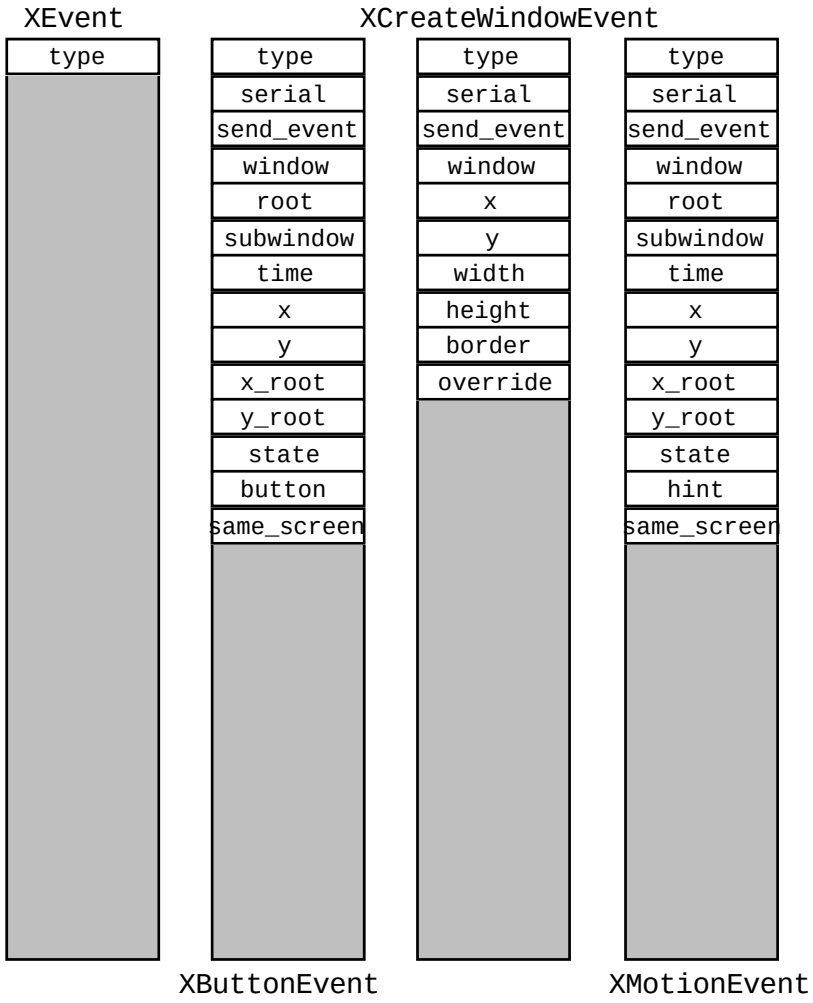

Figure 16.1 XEvents – an example of a union from the Xlib library.

If you have an `XEvent` variable `ev`, you can access its type field using the "." operator just like accessing a data field in a normal structure:

```
XEvent      ev;
…
switch(ev.type) {
    …
}
```

Doubly qualified names to access data members of variants in union

If you know that ev really encodes an XCreateWindowEvent and you want to work out the area of the new window, you can use code like:

```
area = ev.xcreatewindow.width * eve.xcreatewindow.height;
```

The appropriate data fields are identified using a doubly qualified name. The variable name is qualified by the name of the union type that is appropriate for the kind of record known to be present (so, for an `XCreateWindowEvent`, you start with `ev.xcreatewindow`). This name is then further qualified by the name of the data member that should be accessed (`ev.xcreatewindow.width`).

A programmer writing the code to deal with such messages knows that a message will start with a type code. The Xlib library has a series of #defined constants, e.g. ButtonPress, DestroyNotify, MotionNotify; the value in the type field of a message will correspond to one of these constants. This allows messages from an Xterminal to be handled as follows:

```
XEvent eV;
…
/* Code that gets calls the Unix OS and
gets details of the next message from the Xterminal
copied into eV */
…
switch(ev.type) {
caseButtonPress:
    /* code doing something depending on where the button was
    pressed, access using xbutton variant from union */

    if((ev.xbutton.x > x_low) && (ev.xbutton.x < x_high) && …

    break;
case MotionNotify:
    /* user has moved pointer, get details of when this
happened
    and decide what to do, access using xmotion variant from
    union */

    thetime = ev.xmotion.time;
    …
```

```
        break;

    case CreateNotify:
        /* We have a new window, code here that looks at where and
        what size, access using xcreatewindow variant of union */
        int hpos = ev.xcreatewindow.x;
        …
        break;

        …
        }
```