8

Structures, Unions, and Enumerations

Form a more perfect Union.
– The people

- Introduction
- Structures
  struct Layout; **struct** Names; Structures and Classes; Structures and Arrays; Type Equivalence; Plain Old Data; Fields
- Unions
  Unions and Classes; Anonymous **union**s
- Enumerations
  enum classes; Plain **enum**s; Unnamed **enum**s
- Advice

8.1 Introduction

The key to effective use of C++ is the definition and use of user-defined types. This chapter introduces the three most primitive variants of the notion of a user-defined type:

- A **struct** (a structure) is a sequence of elements (called *members*) of arbitrary types.
- A **union** is a **struct** that holds the value of just one of its elements at any one time.
- An **enum** (an enumeration) is a type with a set of named constants (called enumerators).
- **enum class** (a scoped enumeration) is an **enum** where the enumerators are within the scope of the enumeration and no implicit conversions to other types are provided.

Variants of these kinds of simple types have existed since the earliest days of C++. They are primarily focused on the representation of data and are the backbone of most C-style programming. The notion of a **struct** as described here is a simple form of a **class** (§3.2, Chapter 16).

8.2 Structures

An array is an aggregate of elements of the same type. In its simplest form, a **struct** is an aggregate of elements of arbitrary types. For example:

```
struct Address {
        const char* name;       // "Jim Dandy"
        int number;             // 61
        const char* street;     // "South St"
        const char* town;       // "New Providence"
        char state[2];          // 'N' 'J'
        const char* zip;        // "07974"
};
```

This defines a type called **Address** consisting of the items you need in order to send mail to someone within the USA. Note the terminating semicolon.

Variables of type **Address** can be declared exactly like other variables, and the individual *members* can be accessed using the **.** (dot) operator. For example:

```
void f()
{
        Address jd;
        jd.name = "Jim Dandy";
        jd.number = 61;
}
```

Variables of **struct** types can be initialized using the **{}** notation (§6.3.5). For example:

```
Address jd = {
        "Jim Dandy",
        61, "South St",
        "New Providence",
        {'N','J'}, "07974"
};
```

Note that **jd.state** could not be initialized by the string **"NJ"**. Strings are terminated by a zero character, **'\0'**, so **"NJ"** has three characters – one more than will fit into **jd.state**. I deliberately use rather low-level types for the members to illustrate how that can be done and what kinds of problems it can cause.

Structures are often accessed through pointers using the **–>** (**struct** pointer dereference) operator. For example:

```
void print_addr(Address* p)
{
        cout << p–>name << '\n'
                << p–>number << ' ' << p–>street << '\n'
                << p–>town << '\n'
                << p–>state[0] << p–>state[1] << ' ' << p–>zip << '\n';
}
```

When **p** is a pointer, **p–>m** is equivalent to **(*p).m**.

Alternatively, a **struct** can be passed by reference and accessed using the **.** (**struct** member access) operator:

```
void print_addr2(const Address& r)
{
        cout << r.name << '\n'
                << r.number << ' ' << r.street << '\n'
                << r.town << '\n'
                << r.state[0] << r.state[1] << ' ' << r.zip << '\n';
}
```

Argument passing is discussed in §12.2.

Objects of structure types can be assigned, passed as function arguments, and returned as the result from a function. For example:

```
Address current;

Address set_current(Address next)
{
        address prev = current;
        current = next;
        return prev;
}
```

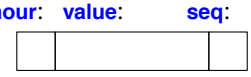
Other plausible operations, such as comparison (**==** and **!=**), are not available by default. However, the user can define such operators (§3.2.1.1, Chapter 18).

8.2.1 **struct** Layout

An object of a **struct** holds its members in the order they are declared. For example, we might store primitive equipment readout in a structure like this:

```
struct Readout {
        char hour;      // [0:23]
        int value;
        char seq;       // sequence mark ['a':'z']
};
```

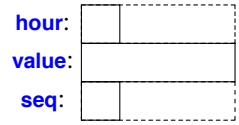
You could imagine the members of a **Readout** object laid out in memory like this:



Members are allocated in memory in declaration order, so the address of **hour** must be less than the address of **value**. See also §8.2.6.

However, the size of an object of a **struct** is not necessarily the sum of the sizes of its members. This is because many machines require objects of certain types to be allocated on architecture-dependent boundaries or handle such objects much more efficiently if they are. For example, integers are often allocated on word boundaries. On such machines, objects are said to have to be properly *aligned* (§6.2.9). This leads to "holes" in the structures. A more realistic layout of a

Readout on a machine with 4-byte **int** would be:

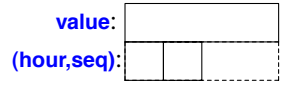


In this case, as on many machines, **sizeof(Readout)** is **12**, and not **6** as one would naively expect from simply adding the sizes of the individual members.

You can minimize wasted space by simply ordering members by size (largest member first). For example:

```
struct Readout {
        int value;
        char hour;      // [0:23]
        char seq;       // sequence mark ['a':'z']
};
```

This would give us:



Note that this still leaves a 2-byte "hole" (unused space) in a **Readout** and **sizeof(Readout)==8**. The reason is that we need to maintain alignment when we put two objects next to each other, say, in an array of **Readout**s. The size of an array of 10 **Readout** objects is **10*sizeof(Readout)**.

It is usually best to order members for readability and sort them by size only if there is a demonstrated need to optimize.

Use of multiple access specifiers (i.e., **public**, **private**, or **protected**) can affect layout (§20.5).

8.2.2 struct Names

The name of a type becomes available for use immediately after it has been encountered and not just after the complete declaration has been seen. For example:

```
struct Link {
        Link* previous;
        Link* successor;
};
```

However, it is not possible to declare new objects of a **struct** until its complete declaration has been seen. For example:

```
struct No_good {
        No_good member;  // error: recursive definition
};
```

This is an error because the compiler is not able to determine the size of **No_good**. To allow two (or

more) structs to refer to each other, we can declare a name to be the name of a struct. For example:

```
struct List;            // struct name declaration: List to be defined later

struct Link {
        Link* pre;
        Link* suc;
        List* member_of;
        int data;
};

struct List {
        Link* head;
};
```

Without the first declaration of List, use of the pointer type List* in the declaration of Link would have been a syntax error.

The name of a struct can be used before the type is defined as long as that use does not require the name of a member or the size of the structure to be known. However, until the completion of the declaration of a struct, that struct is an incomplete type. For example:

```
struct S;  // "S" is the name of some type

extern S a;
S f();
void g(S);
S* h(S*);
```

However, many such declarations cannot be used unless the type S is defined:

```
void k(S* p)
{
        S a;            // error: S not defined; size needed to allocate

        f();            // error: S not defined; size needed to return value
        g(a);           // error: S not defined; size needed to pass argument
        p->m = 7;       // error: S not defined; member name not known

        S* q = h(p);    // ok: pointers can be allocated and passed
        q->m = 7;       // error: S not defined; member name not known
}
```

For reasons that reach into the prehistory of C, it is possible to declare a struct and a non-struct with the same name in the same scope. For example:

```
struct stat { /* ... */ };
int stat(char* name, struct stat* buf);
```

In that case, the plain name (stat) is the name of the non-struct, and the struct must be referred to with the prefix struct. Similarly, the keywords class, union (§8.3), and enum (§8.4) can be used as prefixes for disambiguation. However, it is best not to overload names to make such explicit disambiguation necessary.

8.2.3 Structures and Classes

A **struct** is simply a **class** where the members are **public** by default. So, a **struct** can have member functions (§2.3.2, Chapter 16). In particular, a **struct** can have constructors. For example:

```
struct Points {
    vector<Point> elem;// must contain at least one Point
    Points(Point p0) { elem.push_back(p0);}
    Points(Point p0, Point p1) { elem.push_back(p0); elem.push_back(p1); }
    // ...
};

Points x0;                      // error: no default constructor
Points x1{ {100,200} };         // one Point
Points x1{ {100,200}, {300,400} };  // two Points
```

You do not need to define a constructor simply to initialize members in order. For example:

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

Point p0;       // danger: uninitialized if in local scope (§6.3.5.1)
Point p1 {};    // default construction: {{},{}}; that is {0.0}
Point p2 {1};   // the second member is default constructed: {1,{}}; that is {1,0}
Point p3 {1,2}; // {1,2}
```

Constructors are needed if you need to reorder arguments, validate arguments, modify arguments, establish invariants (§2.4.3.2, §13.4), etc. For example:

```
struct Address {
    string name;        // "Jim Dandy"
    int number;         // 61
    string street;      // "South St"
    string town;        // "New Providence"
    char state[2];      // 'N' 'J'
    char zip[5];        // 07974

    Address(const string n, int nu, const string& s, const string& t, const string& st, int z);
};
```

Here, I added a constructor to ensure that every member was initialized and to allow me to use a **string** and an **int** for the postal code, rather than fiddling with individual characters. For example:

```
Address jd = {
    "Jim Dandy",
    61, "South St",
    "New Providence",
    "NJ", 7974              // (07974 would be octal; §6.2.4.1)
};
```

The **Address** constructor might be defined like this:

```
Address::Address(const string& n, int nu, const string& s, const string& t, const string& st, int z)
                        // validate postal code
        :name{n},
         number{nu},
         street{s},
         town{t}
{
        if (st.size()!=2)
                error("State abbreviation should be two characters")
        state = {st[0],st[1]};          // store postal code as characters
        ostringstream ost;              // an output string stream; see §38.4.2
        ost << z;                       // extract characters from int
        string zi {ost.str()};
        switch (zi.size()) {
        case 5:
                zip = {zi[0], zi[1], zi[2], zi[3], zi[4]};
                break;
        case 4:     // starts with '0'
                zip = {'0', zi[0], zi[1], zi[2], zi[3]};
                break;
        default:
                error("unexpected ZIP code format");
        }
        // ... check that the code makes sense ...
}
```

8.2.4 Structures and Arrays

Naturally, we can have arrays of **struct**s and **struct**s containing arrays. For example:

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

Point points[3] {{1,2},{3,4},{5,6}};
int x2 = points[2].x;

struct Array {
        Point elem[3];
};

Array points2 {{1,2},{3,4},{5,6}};
int y2 = points2.elem[2].y;
```

Placing a built-in array in a **struct** allows us to treat that array as an object: we can copy the **struct** containing it in initialization (including argument passing and function return) and assignment. For example:

```
Array shift(Array a, Point p)
{
        for (int i=0; i!=3; ++i) {
                a.elem[i].x += p.x;
                a.elem[i].y += p.y;
        }
        return a;
}

Array ax = shift(points2,{10,20});
```

The notation for **Array** is a bit primitive: Why **i!=3**? Why keep repeating **.elem[i]**? Why just elements of type **Point**? The standard library provides **std::array** (§34.2.1) as a more complete and elegant development of the idea of a fixed-size array as a **struct**:

```
template<typename T, size_t N >
struct array {    // simplified (see §34.2.1)
        T elem[N];

        T* begin() noexcept { return elem; }
        const T* begin() const noexcept {return elem; }
        T* end() noexcept { return elem+N; }
        const T* end() const noexcept { return elem+N; }

        constexpr size_t size() noexcept;

        T& operator[](size_t n) { return elem[n]; }
        const T& operator[](size_type n) const { return elem[n]; }

        T * data() noexcept { return elem; }
        const T * data() const noexcept { return elem; }

        // ...
};
```

This **array** is a template to allow arbitrary numbers of elements of arbitrary types. It also deals directly with the possibility of exceptions (§13.5.1.1) and **const** objects (§16.2.9.1). Using **array**, we can now write:

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

using Array = array<Point,3>; // array of 3 Points

Array points {{1,2},{3,4},{5,6}};
int x2 = points[2].x;
int y2 = points[2].y;
```

```
Array shift(Array a, Point p)
{
        for (int i=0; i!=a.size(); ++i) {
                a[i].x += p.x;
                a[i].y += p.y;
        }
        return a;
}

Array ax = shift(points,{10,20});
```

The main advantages of **std::array** over a built-in array are that it is a proper object type (has assignment, etc.) and does not implicitly convert to a pointer to an individual element:

```
ostream& operator<<(ostream& os, Point p)
{
        cout << '{' << p[i].x << ',' << p[i].y << '}';
}

void print(Point a[],int s)  // must specify number of elements
{
        for (int i=0; i!=s; ++i)
                cout << a[i] << '\n';
}

template<typename T, int N>
void print(array<T,N>& a)
{
        for (int i=0; i!=a.size(); ++i)
                cout << a[i] << '\n';
}

Point point1[] = {{1,2},{3,4},{5,6}};              // 3 elements
array<Point,3> point2 = {{1,2},{3,4},{5,6}};       // 3 elements

void f()
{
        print(point1,4);        // 4 is a bad error
        print(point2);
}
```

The disadvantage of **std::array** compared to a built-in array is that we can't deduce the number of elements from the length of the initializer:

```
Point point1[] = {{1,2},{3,4},{5,6}};              // 3 elements
array<Point,3> point2 = {{1,2},{3,4},{5,6}};       // 3 elements
array<Point> point3 = {{1,2},{3,4},{5,6}};         // error: number of elements not given
```

8.2.5 Type Equivalence

Two **struct**s are different types even when they have the same members. For example:

```
struct S1 { int a; };
struct S2 { int a; };
```

S1 and **S2** are two different types, so:

```
S1 x;
S2 y = x;  // error: type mismatch
```

A **struct** is also a different type from a type used as a member. For example:

```
S1 x;
int i = x;   // error: type mismatch
```

Every **struct** must have a unique definition in a program (§15.2.3).

8.2.6 Plain Old Data

Sometimes, we want to treat an object as just "plain old data" (a contiguous sequence of bytes in memory) and not worry about more advanced semantic notions, such as run-time polymorphism (§3.2.3, §20.3.2), user-defined copy semantics (§3.3, §17.5), etc. Often, the reason for doing so is to be able to move objects around in the most efficient way the hardware is capable of. For example, copying a 100-element array using 100 calls of a copy constructor is unlikely to be as fast as calling **std::memcpy()**, which typically simply uses a block-move machine instruction. Even if the constructor is inlined, it could be hard for an optimizer to discover this optimization. Such "tricks" are not uncommon, and are important, in implementations of containers, such as **vector**, and in low-level I/O routines. They are unnecessary and should be avoided in higher-level code.

So, a *POD* ("Plain Old Data") is an object that can be manipulated as "just data" without worrying about complications of class layouts or user-defined semantics for construction, copy, and move. For example:

```
struct S0 { };                                  // a POD
struct S1 { int a; };                           // a POD
struct S2 { int a; S2(int aa) : a(aa) { } };    // not a POD (no default constructor)
struct S3 { int a; S3(int aa) : a(aa) { } S3() {} };   // a POD (user-defined default constructor)
struct S4 { int a; S4(int aa) : a(aa) { } S4() = default; };   // a POD
struct S5 { virtual void f(); /* ... */ };      // not a POD (has a virtual function)

struct S6 : S1 { };             // a POD
struct S7 : S0 { int b; };      // a POD
struct S8 : S1 { int b; };      // not a POD (data in both S1 and S8)
struct S9 : S0, S1 {};          // a POD
```

For us to manipulate an object as "just data" (as a POD), the object must

- not have a complicated layout (e.g., with a **vptr**; (§3.2.3, §20.3.2),
- not have nonstandard (user-defined) copy semantics, and
- have a trivial default constructor.

Obviously, we need to be precise about the definition of POD so that we only use such

Entry2::operator=(). Assignment combines the complexities of reading and writing but is otherwise logically similar to the access functions:

```
Entry2& Entry2::operator=(const Entry2& e)  // necessary because of  the string variant
{
    if (type==Tag::text && e.type==Tag::text) {
        s = e.s;          // usual string assignment
        return *this;
    }

    if (type==Tag::text) s.~string(); // explicit destroy (§11.2.4)

    switch (e.type) {
    case Tag::number:
        i = e.i;
        break;
    case Tag::text:
        new(&s)(e.s);   // placement new: explicit construct (§11.2.4)
        type = e.type;
    }

    return *this;
}
```

Constructors and a move assignment can be defined similarly as needed. We need at least a constructor or two to establish the correspondence between the **type** tag and a value. The destructor must handle the **string** case:

```
Entry2::~Entry2()
{
    if (type==Tag::text) s.~string(); // explicit destroy (§11.2.4)
}
```

8.4 Enumerations

An *enumeration* is a type that can hold a set of integer values specified by the user (§iso.7.2). Some of an enumeration's possible values are named and called *enumerators*. For example:

```
enum class Color { red, green, blue };
```

This defines an enumeration called **Color** with the enumerators **red**, **green**, and **blue**. "An enumeration" is colloquially shortened to "an **enum**."

There are two kinds of enumerations:

[1] **enum class**es, for which the enumerator names (e.g., **red**) are local to the **enum** and their values do not implicitly convert to other types

[2] "Plain **enum**s," for which the enumerator names are in the same scope as the **enum** and their values implicitly convert to integers

In general, prefer the **enum class**es because they cause fewer surprises.

8.4.1 enum classes

An **enum class** is a scoped and strongly typed enumeration. For example:

```
enum class Traffic_light { red, yellow, green };
enum class Warning { green, yellow, orange, red };// fire alert levels

Warning a1 = 7;                 // error: no int->Warning conversion
int a2 = green;                 // error: green not in scope
int a3 = Warning::green;        // error: no Warning->int conversion
Warning a4 = Warning::green;    // OK

void f(Traffic_light x)
{
    if (x == 9) { /* ... */ }                   // error: 9 is not a Traffic_light
    if (x == red) { /* ... */ }                 // error: no red in scope
    if (x == Warning::red)  { /* ... */ }       // error: x is not a Warning
    if (x == Traffic_light::red) { /* ... */ }  // OK
}
```

Note that the enumerators present in both **enum**s do not clash because each is in the scope of its own **enum class**.

An enumeration is represented by some integer type and each enumerator by some integer value. We call the type used to represent an enumeration its *underlying type*. The underlying type must be one of the signed or unsigned integer types (§6.2.4); the default is **int**. We could be explicit about that:

```
enum class Warning : int { green, yellow, orange, red }; // sizeof(Warning)==sizeof(int)
```

If we considered that too wasteful of space, we could instead use a **char**:

```
enum class Warning : char { green, yellow, orange, red };     // sizeof(Warning)==1
```

By default, enumerator values are assigned increasing from **0**. Here, we get:

```
static_cast<int>(Warning::green)==0
static_cast<int>(Warning::yellow)==1
static_cast<int>(Warning::orange)==2
static_cast<int>(Warning::red)==3
```

Declaring a variable **Warning** instead of plain **int** can give both the user and the compiler a hint as to the intended use. For example:

```
void f(Warning key)
{
    switch (key) {
    case Warning::green:
        // do something
        break;
    case Warning::orange:
        // do something
        break;
```

```
        case Warning::red:
            // do something
            break;
        }
}
```

A human might notice that `yellow` was missing, and a compiler might issue a warning because only three out of four `Warning` values are handled.

An enumerator can be initialized by a constant expression (§10.4) of integral type (§6.2.1). For example:

```
enum class Printer_flags {
        acknowledge=1,
        paper_empty=2,
        busy=4,
        out_of_black=8,
        out_of_color=16,
        //
};
```

The values for the `Printer_flags` enumerators are chosen so that they can be combined by bitwise operations. An `enum` is a user-defined type, so we can define the `|` and `&` operators for it (§3.2.1.1, Chapter 18). For example:

```
constexpr Printer_flags operator|(Printer_flags a, Printer_flags b)
{
        return static_cast<Printer_flags>(static_cast<int>(a)|static_cast<int>(b));
}

constexpr Printer_flags operator&(Printer_flags a, Printer_flags b)
{
        return static_cast<Printer_flags>(static_cast<int>(a)&static_cast<int>(b));
}
```

The explicit conversions are necessary because a `class enum` does not support implicit conversions. Given these definitions of `|` and `&` for `Printer_flags`, we can write:

```
void try_to_print(Printer_flags x)
{
        if (x&Printer_flags::acknowledge) {
            // ...
        }
        else if (x&Printer_flags::busy) {
            // ...
        }
        else if (x&(Printer_flags::out_of_black|Printer_flags::out_of_color)) {
            // either we are out of black or we are out of color
            // ...
        }
        // ...
}
```

I defined operator|() and operator&() to be constexpr functions (§10.4, §12.1.6) because someone might want to use those operators in constant expressions. For example:

```
void g(Printer_flags x)
{
    switch (x) {
    case Printer_flags::acknowledge:
        // ...
        break;
    case Printer_flags::busy:
        // ...
        break;
    case Printer_flags::out_of_black:
        // ...
        break;
    case Printer_flags::out_of_color:
        // ...
        break;
    case Printer_flags::out_of_black&Printer_flags::out_of_color:
        // we are out of black *and* out of color
        // ...
        break;
    }

    // ...
}
```

It is possible to declare an enum class without defining it (§6.3) until later. For example:

```
enum class Color_code : char;       // declaration
void foobar(Color_code* p);         // use of declaration
// ...
enum class Color_code : char {      // definition
    red, yellow, green, blue
};
```

A value of integral type may be explicitly converted to an enumeration type. The result of such a conversion is undefined unless the value is within the range of the enumeration's underlying type. For example:

```
enum class Flag : char{ x=1, y=2, z=4, e=8 };

Flag f0 {};                         // f0 gets the default value 0
Flag f1 = 5;                        // type error: 5 is not of type Flag
Flag f2 = Flag{5};                  // error: no narrowing conversion to an enum class
Flag f3 = static_cast<Flag>(5);     // brute force
Flag f4 = static_cast<Flag>(999);   // error: 999 is not a char value (maybe not caught)
```

The last assignments show why there is no implicit conversion from an integer to an enumeration; most integer values do not have a representation in a particular enumeration.

Each enumerator has an integer value. We can extract that value explicitly. For example:

```
int i = static_cast<int>(Flag::y);          // i becomes 2
char c = static_cast<char>(Flag::e);        // c becomes 8
```

The notion of a range of values for an enumeration differs from the enumeration notion in the Pascal family of languages. However, bit-manipulation examples that require values outside the set of enumerators to be well defined (e.g., the **Printer_flags** example) have a long history in C and C++.

The **sizeof** an **enum class** is the **sizeof** of its underlying type. In particular, if the underlying type is not explicitly specified, the size is **sizeof(int)**.

8.4.2 Plain enums

A "plain **enum**" is roughly what C++ offered before the **enum class**es were introduced, so you'll find them in lots of C and C++98-style code. The enumerators of a plain **enum** are exported into the **enum**'s scope, and they implicitly convert to values of some integer type. Consider the examples from §8.4.1 with the "**class**" removed:

```
enum Traffic_light { red, yellow, green };
enum Warning { green, yellow, orange, red }; // fire alert levels

// error: two definitions of yellow (to the same value)
// error: two definitions of red (to different values)

Warning a1 = 7;                 // error: no int->Warning conversion
int a2 = green;                 // OK: green is in scope and converts to int
int a3 = Warning::green;        // OK: Warning->int conversion
Warning a4 = Warning::green;    // OK

void f(Traffic_light x)
{
    if (x == 9) { /* ... */ }                   // OK (but Traffic_light doesn't have a 9)
    if (x == red) { /* ... */ }                 // error: two reds in scope
    if (x == Warning::red)  { /* ... */ }       // OK (Ouch!)
    if (x == Traffic_light::red) { /* ... */ } // OK
}
```

We were "lucky" that defining **red** in two plain enumerations in a single scope saved us from hard-to-spot errors. Consider "cleaning up" the plain **enum**s by disambiguating the enumerators (as is easily done in a small program but can be done only with great difficulty in a large one):

```
enum Traffic_light { tl_red, tl_yellow, tl_green };
enum Warning { green, yellow, orange, red }; // fire alert levels
```

```
void f(Traffic_light x)
{
    if (x == red) { /* ... */ }                    // OK (ouch!)
    if (x == Warning::red)  { /* ... */ }          // OK (ouch!)
    if (x == Traffic_light::red) { /* ... */ }     // error: red is not a Traffic_light value
}
```

The compiler accepts the `x==red`, which is almost certainly a bug. The injection of names into an enclosing scope (as `enum`s, but not `enum class`es or `class`es, do) is *namespace pollution* and can be a major problem in larger programs (Chapter 14).

You can specify the underlying type of a plain enumeration, just as you can for `enum class`es. If you do, you can declare the enumerations without defining them until later. For example:

```
enum Traffic_light : char { tl_red, tl_yellow, tl_green };   // underlying type is char

enum Color_code : char;       // declaration
void foobar(Color_code* p);   // use of declaration
// ...
enum Color_code : char { red, yellow, green, blue }; // definition
```

If you don't specify the underlying type, you can't declare the `enum` without defining it, and its underlying type is determined by a relatively complicated algorithm: when all enumerators are non-negative, the range of the enumeration is $[0{:}2^k\text{-}1]$ where 2^k is the smallest power of 2 for which all enumerators are within the range. If there are negative enumerators, the range is $[-2^k{:}2^k\text{-}1]$. This defines the smallest bit-field capable of holding the enumerator values using the conventional two's complement representation. For example:

```
enum E1 { dark, light };                    // range 0:1
enum E2 { a = 3, b = 9 };                   // range 0:15
enum E3 { min = –10, max = 1000000 };       // range -1048576:1048575
```

The rule for explicit conversion of an integer to a plain `enum` is the same as for the `class enum` except that when there is no explicit underlying type, the result of such a conversion is undefined unless the value is within the range of the enumeration. For example:

```
enum Flag { x=1, y=2, z=4, e=8 };    // range 0:15

Flag f0 {};                          // f0 gets the default value 0
Flag f1 = 5;                         // type error: 5 is not of type Flag
Flag f2 = Flag{5};                   // error: no explicit conversion from int to Flag
Flag f2 = static_cast<Flag>(5);      // OK: 5 is within the range of Flag
Flag f3 = static_cast<Flag>(zle);    // OK: 12 is within the range of Flag
Flag f4 = static_cast<Flag>(99);     // undefined: 99 is not within the range of Flag
```

Because there is an implicit conversion from a plain `enum` to its underlying type, we don't need to define `|` to make this example work: `z` and `e` are converted to `int` so that `zle` can be evaluated. The `sizeof` an enumeration is the `sizeof` its underlying type. If the underlying type isn't explicitly specified, it is some integral type that can hold its range and not larger than `sizeof(int)`, unless an enumerator cannot be represented as an `int` or as an `unsigned int`. For example, `sizeof(e1)` could be `1` or maybe `4` but not `8` on a machine where `sizeof(int)==4`.

8.4.3 Unnamed enums

A plain **enum** can be unnamed. For example:

```
enum { arrow_up=1, arrow_down, arrow_sideways };
```

We use that when all we need is a set of integer constants, rather than a type to use for variables.

8.5 Advice

[1] When compactness of data is important, lay out structure data members with larger members before smaller ones; §8.2.1.
[2] Use bit-fields to represent hardware-imposed data layouts; §8.2.7.
[3] Don't naively try to optimize memory consumption by packing several values into a single byte; §8.2.7.
[4] Use **union**s to save space (represent alternatives) and never for type conversion; §8.3.
[5] Use enumerations to represent sets of named constants; §8.4.
[6] Prefer **class enum**s over "plain" **enum**s to minimize surprises; §8.4.
[7] Define operations on enumerations for safe and simple use; §8.4.1.