

Chapter 12

Complex Datatypes

12.1 Custom Types using typedef

12.1.1 Basic usage

- `typedef` can be used to create aliases for types
 - Useful for shortening complex typenames
 - Syntax: `typedef <type> <alias>;`
- Often, new type names are created with suffix `_t`
 - Easier to distinguish them from variable names.
- `typedef` allows to later change types with minimum code changes

12.1.2 Example 1

```
typedef int num_t;

num_t i;
num_t k;
```

- By changing `typedef int num_t` to `typedef long num_t`, `i` and `k` become long values.

12.1.3 Example 2

- `typedef` of a pointer

```
typedef int num_t;

num_t n = 4; // num_t is now an alias for int

typedef num_t* num_ptr; // num_ptr is now a pointer to a num
num_ptr p = &n;
```

12.1.4 typedef for arrays

- Definition of arrays
 - Type of cells is prefix
 - Amount of elements inside `[]` is suffix

```
int array[6];
```

- In typedef same order

```
typedef int i3_t[3];
i3_t array;
array[0] = 3; array[1] = 5; array[2] = 7;
```

12.1.5 Typedef of function pointer

- Similar to arrays
- Return value is a prefix, arguments are appended inside ()

```
typedef int (*binaryfun_t)(int, int);

int plus(int a, int b) { return a + b; }

binaryfun_t fun = plus;
```

12.2 Type Qualifiers

- Type qualifiers annotate a type with further information, thus resulting in a *qualified* type.
 - **unsigned**: The following type does not use its signed bit
 - * Example: **unsigned int i**: A positive integer
 - **const**: After the initialization, no assignment is possible
 - * Example: **const int i = 5**;
 - **volatile**: Type should not be cached (reduces performance but necessary in multithreaded programs).
 - ...

12.3 typedef and Type Qualifiers

- In C, some type qualifiers are very common
 - **unsigned int** usually used for index or size of arrays.
 - Many predefined types using typedef, eg, **uint_t**
 - Suffix **_t** commonly used to not confuse types with variables

12.4 Enumerations

12.4.1 Enumerated Type

- Enumerated type is a type that contains explicitly named constants.
 - Example: Months, days of the week, booleans
 - **enum bool { false, true };**
- Each constant is internally represented by an integer value
 - Values start with 0 and are incremented by 1
 - In the example **bool**, **false** is therefore 0, **true** is 1.

12.4.2 Example

```
enum weekdays {
    sunday, monday, tuesday, wednesday,
    thursday, friday, saturday
};

enum weekdays day = monday;

printf("%d\n", day); // output?
```

12.4.3 Syntax of enum declarations

- Enum declarations can also be used to declare variables

```
enum bool { false, true } b1, b2;

b1 = true;
b2 = false;
```

- Therefore, `enum` declarations must be terminated by a `;`

12.4.4 Custom values in an enumeration

- It a constant in an enumeration should contain a different integer, values can be assigned using `=`.
- Consecutive values are incremented by 1.

12.4.5 Example

- The first month, january, is usually represented by the number 1.

```
enum month {
    january = 1, february, march,
    april, may, june,
    july, august, september,
    october, november, december
};

enum month m = april;

printf("%d\n", m); // output?
```

12.4.6 Example: TCP Ports

- Operating systems use port numbers in TCP connections to specify the service.

```
enum tcp_port {
    ssh = 22, smpt = 25, http = 80, https = 443 };
```

12.4.7 typedef and enums

- By combining `enum` and `typedef` the enum and enum name prefix can be skipped in variable declarations.

```
typedef enum { false, true } bool_t;

bool_t b = false;
```

12.5 Structures

12.5.1 Types so far

- A type is a classification of data (also often called class).
- Examples of types
 - Base types
 - * `char`, `int`, `float`, `double`...
 - Qualified types
 - * `unsigned int`...
 - Pointers
 - * `int *`, `char *`...
 - Arrays
 - * `char s[10]`, `int a[3][10]`, ...
 - Function pointers
 - * `int (*fun)(int)`, ...

12.5.2 Combined types

- Often, data is combined from multiple elements (also called *members*)
- Example: Bot in a game can consist of
 - a name
 - a position
 - * The position itself is represented by two integers (x and y).

12.5.3 Structs

- **structs**: Struct declarations group various elements in one type
- Example: Struct Point
 - Two members: `x` and `y`.

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

- Similar to enumerations, variables can be declared immediately, therefore structures must be terminated with a `;`.
- Order of elements corresponds to order in memory.

12.5.4 Using structures

- Single elements can be accessed using `.`
 - `.` returns an L value

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

```
int main() {
    struct point p;
    p.x = 1; p.y = 2;
    printf("%d %d\n", p.x, p.y);
}
```

12.5.5 Structures and typedef

- Similar to `enum`, `typedef` can be used for a more compact notation.

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

point_t p;
p.x = 3;
p.y = 4;
```

12.5.6 Initialization of Structures

- Structs can be initialized using `{}` (similar to arrays).
- Order of elements corresponds to the order in the `struct` declaration.
- Alternative: Provide name of element.

```
struct point p = { 1, 2 };
struct point q = { .x = 1, .y = 2 };
```

12.5.7 Using Structures in Procedures

- C always uses call-by-value

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

void foo(struct point q) { q.x++; }

int main() {
    struct point p = {1, 2};
    foo(p);
    printf("%d\n", p.x); // output?
}
```

12.5.8 Pointer to Structures

- Structures may be very large
- Passing structures by value can be a bottleneck for memory and efficiency
- Therefore, structures are usually passed as references
 - In Java or C# only way to pass structures to a procedure
- Pointers to structures are very frequent
 - If structure should not be changed, use `const`!
- In C/C++ shortcut for `(*name).member`
 - `name->member`

12.5.9 Example

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

void move(struct point *pp) {
    pp->x = 5;
}

int sum(const struct point *pp) {
    return pp->x + pp->y;
}
```

12.5.10 Exercise

- Write a small collection of geometry procedures.
 - Define a structure for points (use doubles)
 - Use it to define a structure for lines (defined by two points).
 - Look up a formula how to calculate an intersection point.
 - Write a procedure that returns the intersection point of two lines.

12.6 Unions

- Unions resemble structs but are fundamentally different
- Members are stored in the same memory segment

```
union int_or_char {
    int i;
    char ch;
}; // either ch or i are set

int_or_char ic;
ic.c = 'A';

printf("%d", ic.i);
```