Computer Programming: Skills & Concepts (CP)
Structured data: typedef, struct, enum

Kenneth Heafield

Monday 7 November 2016

Last lecture

- Strings

Basic data types in C

int char float double

Really that’s all …
except for variations such as signed char, unsigned char, short, …
- These are the basic options we have for variables.
- We can apply operators to them, compare them etc *, +, ==, < etc.

Today and tomorrow

- typedef – for very simple type definitions.
- struct – for interesting type definitions.
- enum – for set types.
- switch/case statement.
typedef – “create your own types”

Create your own types.

- Well, really just rename the standard ones.
- Use the type just like you would the standard one.
- Useful, for example, in physics:
  Can create metres, kilograms, seconds, joules etc by typedef-ing double. (Unfortunately, C will still let you assign seconds to metres...)

Adding two complex numbers

```
/* i3 and r3 are returned as the result */
int add(double i1, double i2, double r1, double r2, double *r3, double *i3) {
    *r3 = r1 + r2;
    *i3 = i1 + i2;
    return EXIT_SUCCESS;
}
```

Yuck.

More ‘complex’ types

Complex numbers.
Consist of a real and an imaginary part.
Special ways of performing algebraic operations.
Need 2 variables to represent each number.
Messy!

Structured data

Two new data structures. Normally use with typedef.

- struct:
  - Allows you to group related data into a single type.
  - Functions can return a struct and hence return multiple items of data.

- enum:
  - Allows you to define a set of data that will be enumerated to an integer.

Naming convention: common to append ‘_t’ to indicate that the name is a type. Other conventions also used.
A complex number definition

```c
/* Complex number type */
typedef struct {
    /* Real and imaginary parts. */
    double re, im;
} complex_t;
```

A function to return a complex number

we access the member data with .⟨member-name⟩

```c
complex_t MakeComplex (double r, double i) {
    complex_t z;
    z.re = r;
    z.im = i;
    return z;
}
```

Complex number functions

```c
complex_t ComplexSum(complex_t z1, complex_t z2) /* Returns the sum of z1 and z2 */ {
    complex_t z;
    z.re = z1.re + z2.re;
    z.im = z1.im + z2.im;
    return z;
}
```

```c
int ComplexEq(complex_t z1, complex_t z2) /* Testing for equality of structs. */ {
    return (z1.re == z2.re) && (z1.im == z2.im);
}
```
Multiply and modulus

complex_t ComplexMultiply(complex_t z1, complex_t z2)
/* Returns product of z1 and z2 */
{
    complex_t z;
    z.re = z1.re*z2.re - z1.im*z2.im;
    z.im = z1.re*z2.im + z1.im*z2.re;
    return z;
}

double Modulus(complex_t z)
{
    return sqrt(z.re*z.re + z.im*z.im);
}

An example of using these

int main(void)
{
    complex_t z,z1,z2;
    z1 = MakeComplex(1.0, -5.0);
    z2 = MakeComplex(3.0, 2.0);
    z = ComplexMultiply(z1, z2);
    printf("The modulus of z is %f\n", Modulus(z));
    if (ComplexEq(z, MakeComplex(13.0, -13.0))) {
        printf("z is equal to 13-13i\n");
    } else {
        printf("z is not equal to 13-13i\n");
    }
    return EXIT_SUCCESS;
}

Nested structs
A struct can include another struct. This is called nesting.
To access a nested struct member

#include "descartes.h"

typedef struct { point_t points[3]; } triangle_t;

triangle_t tri;
int x_pos = 10;

tri.points[0].x = x_pos;

Because of influences from more modern languages, some would say that
nested access is bad style, and it’s better to write
point_t p0 = tri.points[0];
p0.x = x_pos;

Certainly if you’re going to write tri.points[0] more than once, it pays
to use a variable for it.

Passing struct to a function
Structs can be passed as values to functions:
func1(c1) {

Since C is call by value, the function cannot change member values in the
original struct.
To pass a struct by call by reference:

    Normalize(complex_t *cptr);
    .
    .
    complex_t c1;
    Normalize(&c1);

In most uses of structs, they are always passed via pointers.
**Structs and pointers**

To access the elements of `*cptr`, we have to write `(*cptr).re` and `(*cptr).im`. This rapidly gets boring to type, and is hard to read. C lets us write `cptr->re` and `cptr->im` instead.

```c
void Normalize(complex_t *cptr) {
    double mod = Modulus(*cptr);
    cptr->re = cptr->re / mod;
    cptr->im = cptr->im / mod;
}
```

Structs often contain not other structs, but pointers to other structs. Then we get ‘pointer chasing’:

```
g->players[north]->num_concealed
```

where `g` is a pointer to a struct whose `players` element is an array of pointers to player structs, and a player struct contains an element `num_concealed`.

**Summary (struct)**

- **typedef** allows you to re-name types: Handy with `struct` and `enum`.
- **struct** allows you to group related data into a single variable:
  - Useful for records of multiple items.
  - Bank accounts – name, address, balance etc.
- Can treat `struct` just like any other type:
  - `return` from functions
  - Arrays of `struct`
  - Nested structures
  - Passing structs to a function.

**enum**

Allows data with integer equivalents to be represented:
- For example months of the year.
- Variables are actually stored as integers.

```c
typedef enum {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC} month_t ;
```

```c
typedef struct {
    int day;
    month_t month;
    int year;
} date_t
```

```c
date_t Today;
Today.day = 8 ; Today.month = NOV ; Today.year = 2004
```

**switch/case statement**

- A multiple branch selection statement.
- Tests the value of an expression against a list of integers or character constants.
- Similar to a set of nested if statements:
  - Except can only test for equality.
  - Neater and more readable.
  - Well suited to testing enumerated types
  - *(not good)* need to break out of the switch.
### switch/case standard usage

```c
switch ((expression)) {  
case (constant-1):    
    ⟨statement-sequence-1⟩;  
    break;  
case (constant-2): /* constants are integers */  
    ⟨statement-sequence-2⟩;  
    break;  
case (constant-3):  
    .  
    .  
    default:  
    ⟨statement-sequence⟩;
}
```

### Combining similar cases

```c
date_t Tomorrow(date_t d) {
    switch (d.month) {
        case JAN: case MAR: case MAY: case JUL: case AUG: case OCT:
            if (d.day == 31) {
                d.day = 1; d.month++;  
            } else { d.day++; }
            break;
        /* Now the 30 day months, then February */
        .  
        .  
        case DEC: /* is special */
            if (d.day == 31) {
                d.day = 1; d.month = JAN; d.year++;  
            } else { d.day++; }
        }
    return d;
}
```

### Function to return the next day

```c
date_t Tomorrow(date_t d) {
    switch (d.month) {
        case JAN: case MAR: case MAY: case JUL: case AUG: case OCT:
            if (d.day == 31) {
                d.day = 1; d.month++;  
            } else { d.day++; }
            break;
        /* Now the other months FEB - NOV ...... */
        .  
        .  
        case DEC: /* is special */
            if (d.day == 31) {
                d.day = 1; d.month = JAN; d.year++;  
            } else { d.day++; }
    }
    return d;
}
```

### Summary

- **enum** allows representation of information with integer equivalence:
  - Months, days etc
  - Items in a stock list.
  - Buttons on a ‘pocket calculator’ application.
  - Similar to a set of nested if statements
  - Useful for processing an enumerated type.
  - For example, processing the key pressed in the calculator.