Structured data: typedef and struct

26th October 2010
Last lecture

- Strings.
- Arrays cont. - basic *pattern matching*.
- Bitwise operations on `int` (on board).
Today

▶ `typedef` - for very simple type definitions.
▶ `struct` - for interesting type definitions.
▶ `switch/case` statement.
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.
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 float.
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!
Adding 2 complex numbers

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

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.
/* Complex number type */

typedef struct {
    /* Real and imaginary parts. */
    float re, im;
} Complex_t;
A function to return a complex number

we access the member data with \langle member-name \rangle

Complex_t MakeComplex (float r, float i)
/* Function to create an item of ‘complex number’ type
with real part r, imaginary part i. */
{
    Complex_t z;
    z.re = r;
    z.im = i;
    return z;
}
struct and typedef

With typedef

typedef struct {
...
} Complex_t;

Complex_t a, b;

Without typedef

struct Complex_t {
...
};

struct Complex_t a, b;
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;
}

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;
}

float Modulus(Complex_t z)
{
    return sqrt(z.re*z.re + z.im*z.im);
}
An example of using these

```c
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;
}
```
int main(void)
{
    Complex_t zarr[3] ;
    zarr[0] = MakeComplex(1.0, -5.0);
    zarr[1] = MakeComplex(3.0, 2.0);
    zarr[2] = ComplexMultiply(zarr[0], zarr[1]);
    printf("The modulus of z is \%f\n", Modulus(zarr[2]));
    if (ComplexEq(zarr[2], MakeComplex(13.0, -13.0))) {
        printf("z is equal to 13-13i\n");
    } else
        printf("z is not equal to 13-13i\n");
    /* This line shows how to access individual components */
    printf("z is %d %d i\n", zarr[2].re, zarr[2].im);
    return EXIT_SUCCESS;
}
Nested structs

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

triangle_t tri;
int x_pos = 10;

tri.points[0].x = x_pos;
Passing struct to a function

Structs are passed by call by value.

    func1(c1) { ... 

The function cannot change member values in the struct. To pass a struct by call by reference:

    func1(Complex_t *c1);

    Complex_t c1;
    func1(&c1);
Passing a struct element to a function

Elements are passed by call by value.

    func1(c1.x) { ... }

To pass a struct element by call by reference:

    func1(int *x);

    Complex_t c1;
    func1(&c1.x);
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.

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

typedef struct {
   int day;
   Month_t month;
   int year;
} Date_t

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 syntax

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⟩
}
Function to return the next day

Date_t Tomorrow(Date_t d) {
    switch (d.month) {
    case JAN:
        if (d.day == 31) {
            d.day = 1; d.month = FEB;
        } else
            d.day += 1;
        break;
    /* Now the other months FEB - NOV ....... */
    ...
    case DEC:
        if (d.day == 31) {
            d.day = 1; d.month = JAN; d.year++;
        } else
            d.day += 1;
    }
    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.

**switch/case** statement:

- Similar to a set of nested if statements
- Useful for processing an enumerated type.
- For example, processing the key pressed in the calculator.