We can use the keyword `typedef` to give a name to a type:

```c
typedef double real;
```

This means variables can be declared as `numeric` but they will actually be of type `double`. Do not overuse `typedef` - it can make programs harder to read, e.g.:

```c
typedef int andrew;

andrew main(void) {
    andrew i,j;
    ....
```
Using typedef to make programs portable

Suppose have a program that does floating-point calculations. If we use a typedef’ed name for all variable, e.g.:

```c
typedef double real;

real matrix[1000][1000][1000];

real my_atanh(real x) {
    real u = (1.0 - x)/(1.0 + x);
    return -0.5 * log(u);
}
```

If we move to a platform with little RAM, we can save memory (and lose precision) by changing the typedef:

```c
typedef float real;
```
• We have seen simple types e.g. `int, char, double`
  ▶ variables of these types hold single values

• We have seen a compound type: arrays
  ▶ array variables hold multiple values
  ▶ arrays are homogenous - every array element is the same type
  ▶ array element selected using integer index
  ▶ array size can be determined at runtime

• Another compound type: structs
  ▶ structs hold multiple values (fields)
  ▶ struct are heterogeneous - fields can be different type
  ▶ struct field selected using name
  ▶ struct fields fixed
If we define a struct that holds COMP1511 student details:

```c
#define MAX_NAME 64
#define N_LABS 10

struct student {
    int zid;
    char name[MAX_NAME];
    double lab_marks[N_LABS];
    double assignment1_mark;
    double assignment2_mark;
};
```

We can declare an array to hold the details of all students:

```c
struct student comp1511_students[900];
```
combining structs and typedef

Common to use typedef to give name to a struct type.

```c
struct student {
    int zid;
    char name[64];
    double lab_marks[N_LABS]
    double assignment1_mark;
    double assignment2_mark;
}

typedef struct student student_t;

student_details_t comp1511_students[900];
```

Programmer often use convention to separate type names e.g. _t suffix.
Assigning structs

Unlike arrays, it is possible to copy all components of a structure in a single assignment:

```c
struct student_details student1, student2;
...
student2 = student1;
```

It is not possible to compare all components with a single comparison:

```c
if (student1 == student2) // NOT allowed!
```

If you want to compare two structures, you need to write a function to compare them component-by-component and decide whether they are “the same”. 
A structure can be passed as a parameter to a function:

```c
void print_student(student_t student) {
    printf("%s z%d\n", d.name, d.zid);
}
```

Unlike arrays, a copy will be made of the entire structure, and only this copy will be passed to the function.

Unlike arrays, a function can return a struct:

```c
student_t read_student_from_file(char filename[]) {
    ...
}
```
Pointers to structs

If a function needs to modify a structs field or if we want to avoid the inefficiency of copying the entire struct, we can instead pass a pointer to the struct as a parameter:

```c
int scan_zid(student *s) {
    return scanf("%d", &((*s).zid));
}
```

The “arrow” operator is more readable:

```c
int scan_zid(student *s) {
    return scanf("%d", &(s->zid));
}
```

If s is a pointer to a struct s->field is equivalent to (*s).field
Nested Structures

One structure can be nested inside another

```c
typedef struct date Date;
typedef struct time Time;
typedef struct speeding Speeding;

struct date {
    int day, month, year;
};
struct time {
    int hour, minute;
};
struct speeding {
    Date date;
    Time time;
    double speed;
    char plate[MAX_PLATE];
};
```