NWEN 241
User Defined Types

Qiang Fu

School of Engineering and Computer Science
Victoria University of Wellington

This Lecture

- More on data types

Data Types

- Basic types: int, char, float, double, void, etc
- Derived types: arrays of basic types, pointers to basic types, and functions returning basic types
- Wouldn't it be nice to build user-defined types.

Renaming Types

- **typedef** declares a new name for a specified type

  `typedef type newname;`

  – For example:

```
typedef int Time;    /* Time is an alias of int */
Time hours, minutes, seconds;
```

  – **typedef** does not define a new type

  – A pointer to a function that returns a pointer to a function that returns a pointer to a char

```
/* char *(*(*)())() */
typedef char *(*(*pfpfpc)())();

pfpfpc a;

/* Or */
```

Renaming Types

- **typedef** declares a new name for a specified type

```
typedef type newname;
```

  - For example:

```
typedef int Time;    /* Time is an alias of int */
Time hours, minutes, seconds;
```

  - **typedef** does not define a new type
  - A pointer to a function that returns a pointer to a function that returns a pointer to a char

```
/* char *(*(*)())() */
typedef char *pc;
typedef pc fpc();
typedef fpc *pfpc;
typedef pfpc fpfpc();
typedef fpfpc *pfpfpc;

pfpfpc a;
```


Enumeration Types

- A simple example of user-defined types
- Enumerated types contain a list of names

```
enum tag {enumerator list};
```

  - For example:

```
enum Colour {Red, Green, Blue, Black} flag;
```

  - Use typedef to rename enum Colour

```
typedef enum Colour Colour;
Colour aflag = Red;


Colour suit = Black;
```


Enumeration Types

- A simple example of user-defined types
- Enumerated types contain a list of names

```
enum tag {enumerator list};
```

  - For example:

```
enum Colour {Red, Green, Blue, Black} flag;
  /* flag is of type enum Colour */
```

  - Use typedef to rename enum Colour

```
typedef enum Colour Colour;
Colour aflag = Red;


Colour suit = Black;
```


Enumeration Types

- A simple example of user-defined types
- Enumerated types contain a list of names

```
enum tag {enumerator list};
```

  - For example:

```
enum Colour {Red, Green, Blue, Black} flag;
  /* flag is of type enum Colour */
```

  - Use typedef to rename enum Colour

```
typedef enum Colour Colour;
Colour aflag = Red;
   /* declare aflag is of type Colour */
   /* and initialise aflag with Red */
Colour suit = Black;
```

Enumeration Types

- What is behind these names

```
enum Colour {Red, Green, Blue};
```


Enumeration Types

- What is behind these names – integer constants

```
enum Colour {Red, Green, Blue};
```

is automatically defined as:

```
enum Colour {Red=0, Green=1, Blue=2};
```


Enumeration Types

- What is behind these names – integer constants

```
enum Colour {Red, Green, Blue};
```

is automatically defined as:

```
enum Colour {Red=0, Green=1, Blue=2};
```

However, we can override the default values

```
enum Colour {Red=10, Green, Blue};
```

```
enum Colour {Red=3, Green=1, Blue=5};
enum Colour {Red=0, Green=0, Blue=0,
  Yellow=3,...};
```


Enumeration Types

- What is behind these names – integer constants

```
enum Colour {Red, Green, Blue};
```

is automatically defined as:

```
enum Colour {Red=0, Green=1, Blue=2};
```

However, we can override the default values

```
enum Colour {Red=10, Green, Blue};
   /* Green is automatically assigned 11 */
   /* Blue is automatically assigned 12 */
enum Colour {Red=3, Green=1, Blue=5};
enum Colour {Red=0, Green=0, Blue=0,
  Yellow=3,...};
```

Enumeration Types

- Make a Boolean type yourself
- enum vs. #define
  - Both provide a way to associate integer constants with names
  - enum can generate values automatically
- Be aware…

Enumeration Types

- Make a Boolean type yourself
- enum vs. #define
  - Both provide a way to associate integer constants with names
  - enum can generate values automatically
- Be aware…
  - Names used in an enumeration cannot be used in another enumeration within the same scope
  - Names must be valid identifiers

Enumeration Types

- Make a Boolean type yourself
- enum vs. #define
  - Both provide a way to associate integer constants with names
  - enum can generate values automatically
- Be aware…
  - Names used in an enumeration cannot be used in another enumeration within the same scope

```
enum Colour {Red, Green, Blue, Orange};
enum Fruit {Apple, Grape, Orange, Pear};
```

  - Names must be valid identifiers

Enumeration Types

- Make a Boolean type yourself
- enum vs. #define
  - Both provide a way to associate integer constants with names
  - enum can generate values automatically
- Be aware…
  - Names used in an enumeration cannot be used in another enumeration within the same scope

```
enum Colour {Red, Green, Blue, Orange};
enum Fruit {Apple, Grape, Orange, Pear};
```

  - Names must be valid identifiers

```
enum Grade {E, D, ..., A-, A+};
```

Enumeration Types

- An example – use three primary colours

```
enum Colour {Red=0, Green=0, Blue=0, ..., Purple};
typedef enum Colour Colour;

Colour c_array[]={Red, Purple, Black, Green, Orange};

int i, nc = sizeof(c_array)/sizeof(c_array[0]);

for (i =0; i<nc; i++) { /* one of the three primary */
    if(!c_array[i])     /* colours? */
      ...;
    else
      ...;
}
```


Type Casting

- We talked about this before
  - Force one variable of one type to be another type

```
(typename)expression;
int i, ii = 5;
float f = 3.14, ff;
i = f;       /* can you do this in java? */
ff = ii;     /* can you do this in java? */
```


Type Casting

- We talked about this before
  - Force one variable of one type to be another type

```
(typename)expression;
int i, ii = 5;
float f = 3.14, ff;
i = (int)f;       /* i=3, f=3.14 or 3.0? */
ff = (float)ii;   /* ff = 5.0 */
```


Type Casting

- We talked about this before
  - Force one variable of one type to be another type

```
(typename)expression;
int i, ii = 5;
float f = 3.14, ff;
i = (int)f;       /* i=3, f=3.14 or 3.0? */
ff = (float)ii;   /* ff = 5.0 */

/* C will do this kind of type casting */
/* automatically, but there are */
/* many cases we have to do */
/* type casting ourselves */
```

Type Casting

- We talked about this before
  - Explicit type casting

```
a = b;  /* if you know a's type, */
        /* but not sure about b's type */;
```


Structures

- Structures vs. arrays
  - Members in an array must be of the same type
  - Members in a struct can be of different types

```
struct tag {member1; . . . member n;};
```

- struct is a simplified version of class
  - A class with only public members and no functions
- A struct template

```
struct Person {
    char *name;        /* name[50]? */
    char gender;
    int age;
  }; /* struct needs a ";" as array does */
```


Structures

- Use typedef to rename struct Person

```
typedef struct Person Person;
```

- Let us declare a couple of Person objects

```
Person bob, sue;

bob.name = "Robert Jackson"; /* name[50]?*/
bob.gender = 'M';
bob.age = 48;

sue.name = "Suzan Jackson";
sue.gender = 'F';
sue.age = 20;
```


Structures

- Nested structures
  - Let us add a new member to Person

```
struct Date {
  int day;
  int month;
  int year;
};
typedef struct Date Date;
/* Or */
```

Structures

- Nested structures
  - Let us add a new member to Person

```
struct Date {
  int day;
  int month;
  int year;
};
typedef struct Date Date;
/* Or */
typedef struct {
  int day;
  int month;
  int year;
} Date;
```


Structures

- Nested structures
  - Let us add a new member to Person

```
struct {
  int day;
  int month;
  int year;
} Date;

/* What is this? */
```


Structures

- Nested structures
  - Let us add a new member to Person

```
struct {
  int day;
  int month;
  int year;
} Date;

/* it is bad .... */
```


Structures

- Nested structures
  - Let us add a new member to Person

```
typedef struct {
  int day;
  int month;
  int year;
} Date;

struct Person {
  char name[50];
  char gender;
  int age;
  Date birthday;
};
```

Structures

- Nested structures
  - Add sue's birthday

```
Date abirthday = {27, 7, 1989};
                        /* initialisation */
sue.birthday = abirthday;
                        /* assignment */
```

  - Can we do:

```
sue.birthday = {27, 7, 1989};
```


Structures

- Be aware…

```
typedef struct Person {
    char *name;        /* name[50]? */
    char gender;
    int age;
  } Person;
```

  - Initialisation

```
Person john = {"John Jackson", 'M', 18,
  {12, 3, 1991}};      /* name[50]? */
```

  - Can we do this assignment?

```
Person john;
john = {"John J", 'M', 18, {12, 3, 1991}};
```


Structures

- Pointers to structures

```
Person *pjohn = &john;

  /* modify john's age */


  /* use john directly */


  /* use a pointer to john */


  /* use a pointer to get john, and then use john */
```


Structures

- Pointers to structures

```
Person *pjohn = &john;

  /* modify john's age */

john.age = 20;
  /* use john directly */


  /* use a pointer to john */


  /* use a pointer to get john, and then use john */
```

Structures

- Pointers to structures

```
Person *pjohn = &john;

  /* modify john's age */

john.age = 20;
  /* use john directly */

pjohn->age = 30;
  /* use a pointer to john */


  /* use a pointer to get john, and then use john */
```


Structures

- Pointers to structures

```
Person *pjohn = &john;

  /* modify john's age */

john.age = 20;
  /* use john directly */

pjohn->age = 30;
  /* use a pointer to john */

(*pjohn).age = 40;
  /* use a pointer to get john, and then use john */
```


Structures

- Be aware…
  - Variables of the same struct type can be assigned by one another

```
struct SWEN201 {
  int year;
  int enrolments;
  char *class_rep;
};
typedef struct SWEN201 SWEN201;

SWEN201 sy09, sy2009 = {2009, 40, "Peter"};

sy09 = sy2009;
```


Structures

- Be aware…
  - How about variables of the similar struct types?

```
struct COMP206 {
  int year;
  int enrolments;
  char *class_rep;
};

typedef struct COMP206 COMP206;

COMP206 cy09, cy2009 = {2009, 60, "John"};

cy09 = sy2009;
sy09 = cy2009;
```

Structures

- Be aware…
  - Variables of the similar struct type cannot

```
struct COMP206 {
  int year;
  int enrolments;
  char *class_rep;
};

typedef struct COMP206 COMP206;

COMP206 cy09, cy2009 = {2009, 60, "John"};

cy09 = sy2009;       /* wrong */
sy09 = cy2009;       /* wrong */
```


Structures

- Be aware…
  - If we insist to mix up SWEN and COMP…

```
typedef struct {     /* no tag name here */
  int year;
  int enrolments;
  char *class_rep;
} COMP206, SWEN201;

COMP206 cy09, cy2009 = {2009, 60, "John"};
SWEN201 sy09, sy2009 = {2009, 40, "Peter"};

cy09 = sy2009;
sy09 = cy2009;
```


Structures

- Be aware…
  - If we insist to mix up SWEN and COMP…

```
Typedef struct {     /* no tag here */
  int year;
  int enrolments;
  char *class_rep;
} COMP206, SWEN201;

COMP206 cy09, cy2009 = {2009, 60, "John"};
SWEN201 sy09, sy2009 = {2009, 40, "Peter"};

cy09 = sy2009;       /* accepted */
sy09 = cy2009;       /* accepted */
```


Structures

- Pointers to structures

```
SWEN201 *psy2009 = &sy2009;
psy2009->enrolments = 40;/* (*psy2009).enrolments */
```

- Structures with pointer members

```
struct Person {
  char *name;
  int *age;
  Date *birthday;
} bill;
...;           /* get bill initialised */
bill.name = “Bill Johnson”;
         /* string constant returns a pointer */
*bill.age = 32; /* any thing wrong? */
bill.birthday->year = 1977; /* any thing wrong? */
```

Structures

- Pointers to structures

```
SWEN201 *psy2009 = &sy2009;
psy2009->enrolments = 40;/* (*psy2009).enrolments */
```

- Structures with pointer members

```
struct Person {
  char *name;
  int *age;
  Date *birthday;
} bill;
...;          /* get bill initialised */
bill.name = “Bill Johnson”;
         /* string constant returns a pointer */
*bill.age = 32; /* “.” is of higher precedence */
bill.birthday->year = 1977; /*associativity L to R*/
```


Passing Structures to Functions

- Is a structure passed to a function by value?

Passing Structures to Functions

- When a structure is passed to a function, it is passed by value
- But, we can also pass the address of the structure to the function

Passing Structures to Functions

- An example (call-by-value vs. call-by-reference)

```
typedef struct Person {
  ...
} Person;
Person john = {...};       /* initialisation */
-----------------------
john = update(john);       /* update john’s info */
Person update(Person aname)
{ ...
  return aname;
}
-----------------------
update(&john);             /* update john’s info */
void update(Person *ptr)
{ ...
}
```

Passing Structures to Functions

- Pass by value vs. pass by address
  - What's good for you???

Passing Structures to Functions

- An example (call-by-value vs. call-by-reference)

```
typedef struct {
  char name[50];
  ...
} Person;
Person john = {“John H”, ...}; /* initialisation */

john = update(john);       /* update john’s info */

void update(Person p)
{
  printf("Printing the old name: %s\n", p.name);
  printf("Type in a new name:\n");
  scanf(" %[^\n]", p.name);    /* “John B” */
}
```


Size of Structures

- Tell me the sizes of the two structures

```
typedef struct Size1 {
  char achar;
  char bchar;
  char cchar;
  char dchar;
  char echar;
  struct Size1 *next;
} Size1;

typedef struct Size2 {
  int aint;
  int bint;
  char achar;
} Size2;
```


Size of Structures

- Tell me the sizes of the two structures

```
typedef struct Size1 {
  char achar;
  char bchar;
  char cchar;
  char dchar;
  char echar;
  struct Size1 *next;
} Size1;                    /* Size1 = 12 */

typedef struct Size2 {
  int aint;
  int bint;
  char achar;
} Size2;                    /* Size2 = 12 */
```

Unions

- Unions vs. structures
  - Unions follows the same syntax as structures
  - The members of unions have to share storage (only one member can have storage at a time)

```
struct int_and_float {
  int i;       /* storage allocated to */
  float f;     /* s_number to accommodate */
} s_number;    /* both i and f */

union int_or_float {
  int i;       /* the storage allocated to */
  float f;     /* u_number can accommodate */
} u_number;    /* the largest number (f) */
u_number.i = 11;   /* no storage for f */
u_number.f = 99.0; /* no storage for i */
```


Unions

- What are unions good for
  - Share the same piece of memory between different types of data
  - Reduce the consumption of memory