Personal SE

C Struct & Typedef

Make

C Structs

- A *struct* is a way of grouping named, heterogeneous data elements that represent a coherent concept.

C Structs

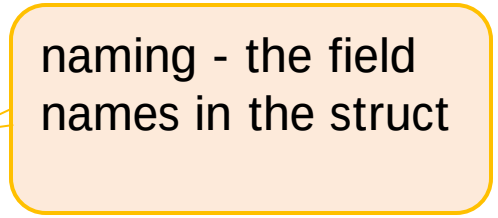
- A struct is a way of grouping named, heterogeneous data elements that represent a coherent concept.
- Example:

```
#define MAXNAME (20)

struct person {
    char name[MAXNAME+1] ;
    int age ;
    double income ;
} ;
```

C Structs

- Question: What is an object with no methods and only instance variables public?
- Answer: A struct! (well, sort of).
- A struct is a way of grouping named, heterogeneous data elements that represent a coherent concept.
- Example:

```
#define MAXNAME (20)

struct person {
    char name[MAXNAME+1] ;
    int age ;
    double income ;
} ;
```



Software Engineering
Rochester Institute of Technology

C Structs

- Question: What is an object with no methods and only instance variables public?
- Answer: A struct! (well, sort of).
- A struct is a way of grouping named, heterogeneous data elements that represent a coherent concept.
- Example:

```
#define MAXNAME (20)

struct person {
    char name[MAXNAME+1] ;
    int age ;
    double income ;
} ;
```

heterogeneous - the fields have different types

C Structs

- Question: What is an object with no methods and only instance variables public?
- Answer: A struct! (well, sort of).
- A struct is a way of grouping named, heterogeneous data elements that represent a coherent concept.
- Example:

```
#define MAXNAME (20)

struct person {
    char name[MAXNAME+1] ;
    int age ;
    double income ;
} ;
```

coherent concept - the information recorded for a person.

Using Structs

- Declaration:

```
struct person {
    char name[MAXNAME+1] ;
    int age ;
    double income ;
} ;
```

- Definitions:

```
struct person mike,
              pete ;
```

- Assignment / field references ('dot' notation):

```
mike = pete ;
pete.age = chris.age + 3
```

Using Structs

- Note: Space allocated for the whole struct at definition.
- Struct arguments are passed by value (i.e., copying)

WRONG

```
void give_raise(struct person p, double pct) {
    p.income *= (1 + pct/100) ;
    return ;     // Note that return is not needed for void function
}

give_raise(mike, 10.0) ;
```

RIGHT

```
struct person give_raise(struct person p, double pct) {
    p.income *= (1 + pct/100) ;
    return p ;   // must return struct person
}

mike = give_raise(mike, 10.0) ;
```

Symbolic Type Names - typedef

- Suppose we have a pricing system that prices goods by weight.
  - Weight is in pounds, and is a double precision number.
  - Price is in dollars, and is a double precision number.
  - Goal: Clearly distinguish weight variables from price variables.

Symbolic Type Names - typedef

- Suppose we have a pricing system that prices goods by weight.
  - Weight is in pounds, and is a double precision number.
  - Price is in dollars, and is a double precision number.
  - Goal: Clearly distinguish weight variables from price variables.
- Typedef to the rescue:
  - typedef ***declaration*** ;Creates a new "type" with the variable slot in the ***declaration***.

Symbolic Type Names - typedef

- Suppose we have a pricing system that prices goods by weight.
  - Weight is in pounds, and is a double precision number.
  - Price is in dollars, and is a double precision number.
  - Goal: Clearly distinguish weight variables from price variables.
- Typedef to the rescue:
  - typedef ***declaration*** ; Creates a new "type" with the variable slot in the ***declaration***. Use a "_t" suffix to identify it as a typedef.
- Examples:

```
typedef double price_t ;     // alias for double to declare price variabless
typedef double weight_t ;    // alias for double to declare weight variables

price_t     p ;              // double precision value that's a price
weight_t  lbs ;              // double precision value that's a weight
```

typedef In Practice

- Symbolic names for array types

```
#define  MAXSTR  (100)

typedef  char  long_string_t[MAXSTR+1] ;

long_string_t  line ;
long_string_t  buffer ;
```

typedef In Practice

- Shorter name for struct types:

```
typedef struct  {
   long_string_t  label ;   // name for the point
   double  x ;              // xcoordinate
   double  y ;              // ycoordinate
}  point_t ;                // pick a name that suggests it is a struct

point_t  origin ;
point_t  focus ;
```

Make and Makefiles

- Problem:
  - Program comprises many source files.

Make and Makefiles

- Problem:
  - Program comprises many source files.
  - Recompiling everything is time-consuming and redundant.

Make and Makefiles

- Problem:
  - Program comprises many source files.
  - Recompiling everything is time-consuming and redundant.
  - Changes to a file may make other files obsolete.

Make and Makefiles

- Problem:
  - Program comprises many source files.
  - Recompiling everything is time-consuming and redundant.
  - Changes to a file may make other files obsolete.
  - How can we periodically regenerate the executable doing the minimum amount of work?

Make and Makefiles

- Problem:
  - Program comprises many source files.
  - Recompiling everything is time-consuming and redundant.
  - Changes to a file may make other files obsolete.
  - How can we periodically regenerate the executable doing the minimum amount of work?
- Solution: *make* (or *ant*, *rake* and other similar programs)

Make and Makefiles

- Problem:
  - Program comprises many source files.
  - Recompiling everything is time-consuming and redundant.
  - Changes to a file may make other files obsolete.
  - How can we periodically regenerate the executable doing the minimum amount of work?
- Solution: *make* (or *ant*, *rake* and other similar programs)
  - Record obsolescence dependencies: a Directed Acyclic Graph (DAG)

Make and Makefiles

- Problem:
  - Program comprises many source files.
  - Recompiling everything is time-consuming and redundant.
  - Changes to a file may make other files obsolete.
  - How can we periodically regenerate the executable doing the minimum amount of work?
- Solution: *make* (or *ant*, *rake* and other similar programs)
  - Record obsolescence dependencies: a Directed Acyclic Graph (DAG)
  - Define commands to recreate obsolete files.

Make and Makefiles

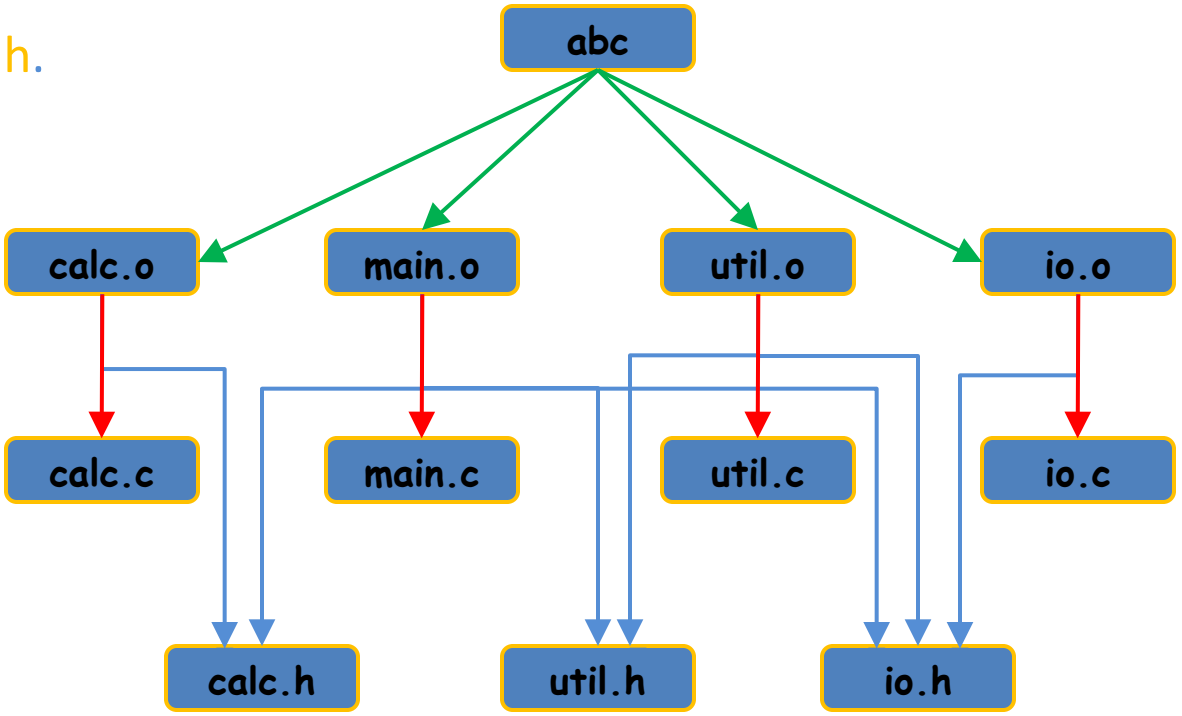
- Problem:
  - Program comprises many source files.
  - Recompiling everything is time-consuming and redundant.
  - Changes to a file may make other files obsolete.
  - How can we periodically regenerate the executable doing the minimum amount of work?
- Solution: *make* (or *ant*, *rake* and other similar programs)
  - Record obsolescence dependencies: a Directed Acyclic Graph (DAG)
  - Define commands to recreate obsolete files.
  - Depth first traversal of the DAG to bring things up-to-date.

What Is A Dependency?

- File *A depends* on file B if the correctness of *A*'s contents are affected by changes to B.
- Thus an object file depends on its source:
  - A change to the source makes the object file incorrect.
- An object file depends on interfaces its source file uses:
  - Interface change may change the meaning of the source code
  - E.g., change a configuration constant, a struct, etc.
- An executable program depends on the object code files from which it is built.

Example

- Program abc made from main.o, util.o, calc.o and io.o.
- main.c includes calc.h, util.h and io.h.
- util.c includes util.h and io.h.
- calc.c includes calc.h.
- io.c includes io.h.



DEPENDENCY KEY

program to object **green**
object to source **orange**
object to interface **blue**

Dependencies in Makefiles

target: $dependency_1$ $dependency_2$ . . . $dependency_N$

For our example the dependency lines are

```
abc: main.o util.o calc.o io.o

main.o: main.c util.h calc.h io.h

util.o: util.c util.h io.h

calc.o: calc.c calc.h

io.o: io.c io.h
```

Is a Target Up-To-Date?

- A target is *up-to-date* iff
  - It exists (obviously).
  - It was modified later than any of its dependencies *after they have all been brought up-to-date*.
- What do we do if a file is *not* up-to-date?
  - We run one or more commands to bring it up-to-date.
  - For a program, we link the object files.
  - For an object file, we recompile its source.
- For make, command lines:
  - Follow the dependency line.
  - ***MUST*** begin with a ***hard tab*** (Tab key or CTRL-I).

Completed Makefile for the Example

```
abc: main.o util.o calc.o io.o
        gcc -o abc –g main.o util.o calc.o io.o

main.o: main.c util.h calc.h io.h
        gcc -c –Wall –g main.c

util.o: util.c util.h io.h
        gcc -c –Wall –g util.c

calc.o: calc.c calc.h
        gcc -c –Wall –g calc.c

io.o: io.c io.h
        gcc -c –Wall –g io.c
```

Assuming Existence of "Makefile"

`make`

- Brings the default up to date which is the first target (abc in this case)

`make abc`

- Specifically brings abc up to date.
- First brings main.o util.o calc.o and io.o up to date
- Then relink abc iff
  - abc does not exist
  - abc is older than at least one of its dependencies (any of four .o files)

`make main.o`

- Just brings main.o up to date.
- Any target can be specified.

Things to Note

- Targets need not have any dependencies.
- Targets need not ever really be made – runs command(s) every time.
- Multiple commands can be run.
- Example: Generic "clean" target:

```
clean:
        rm -f *.o *~* abc
```