C Structs

Kurt Schmidt
Skipjack Solutions

November 8, 2021
Notes

Intended audience: Student who has working knowledge of Python

Target compiler: I’ll try to center the discussion on C99 using gcc 7.4

- Code examples might have an accompanying link
  - Follow link to step through example at pythontutor.com
  - Does a nice job of graphically showing variables in memory, the heap, and the stack

- In the interest of clarity, avoiding clutter, many examples lack a NOMEM check
  - *Never* assume malloc, etc., succeeded
Intro
C Structs – Aggregate Types

User-defined type
- Type name is `struct foo`
- This is a common idiom

```c
typedef struct foo foo ;
```

Collection of fields
- Field can be any defined type
- Fields are accessed with a dot (.)
  - The member-access operator

```c
struct foo {  
  int i ;  
  char c ;  
  float x ;  
  char name[10] ;  
} ;
```
#include <string.h>
#include <stdio.h>

struct person {
    int age ;
    char id[10] ; /* string */
};

int main()
{
    struct person p ;
    p.age = 28;
    strcpy( p.id, "abc123" ) ;

    printf( "%s has %d years.\n", p.id, p.age ) ;

    return 0 ;
}

$ gcc structs.c -o structs
$ ./structs
abc123 has 28 years.

- Type definitions (e.g., structs) end w/a semicolon
- Functions do not
- Do not assign (=) strings
- Use strcpy
We can initialise a struct object when it’s declared
Initialiser are positional only
  Must be in the same order

```c
struct person {
    int age ;
    double mass ;
    char id[10] ; /* string */
};

int main()
{
    struct person p = { 27, 106.7, "Kurt" } ;

    printf( "%s is %d years old, with a mass of %f kg..\n",
            p.id, p.age, p.mass ) ;

    return 0 ;
}
```
typedef
typedef

typedef <existing type> <new name>

• Creates aliases for existing types
• E.g.:

```c
typedef double coord_t;
typedef unsigned long size_t;
typedef struct Person Person;
typedef struct sNode* Stack;
```
We can combine a `typedef` with an *unnamed* type:

```c
typedef struct {
    int age;
    char id[10]; /* string */
} person;

int main()
{
    person p;
    ...
}
```
Pointers to
Pointers to Structs

- We can ask for a pointer to any variable:

```c
person p;
person *pp = &p;
```

- If `pp` points to a `person`, then `*pp` is that person

- We can access fields:

```c
*pp.age = 27;  /* NO! */
```

- `.` has higher precedence than `*`, so:

```c
(*pp).age = 27;  /* Correct */
```

- Clunky. But, we have syntactic sugar:

```c
pp->age = 27;  /* Prettier */
```
We can get a `person` from the heap:

```c
typedef struct {
    int age;
    char id[10]; /* string */
} person;

int main() {
    person *p = (person*) malloc( sizeof( person ));
    if( p==NULL ) ... ;

    strcpy( p->id, "THX1138" );
    p->age = 23;

    // ...

    free( p );
    p = NULL;

    return 0;
}
```
Arrays
typedef struct
{
    int age ;
    char id[10] ; /* string */
} person ;

int main()
{
    person p[20] ;

    p[3].age = 42 ;
    strcpy( p[3].id, "THX1138" ) ;

    return 0 ;
}
```c
typedef struct {
    int age ;
    char id[10] ; /* string */
} person ;

enum { MAX=12 } ;

int main() {
    person* team[MAX] = { NULL } ; // Array of Pointers

team[0] = (person*) malloc( sizeof( person ) ) ;
assert( team[0] != NULL ) ;
team[0]->age = 36 ;
strcpy( team[0]->id, "Bashful" ) ;

    // free the people!
    return 0 ;
}
```

[0]https://tinyurl.com/y5l3pper
int main() {
    size_t size = 12;
    person** team; // Pointer to person ptr (array of person*)

    team = (person**) malloc( size * sizeof( person* ) );
    for( int i=0; i<size; ++i )
        team[i] = NULL;

    team[0] = (person*) malloc( sizeof( person ) );
    team[0]->age = 36;
    strcpy( team[0]->id, "Bashful" );
    // ...

    for( int i=0; i<size && team[i]!=NULL; ++i ) { // Assume dense array
        free( team[i] );
        team[i] = NULL;
    }
    free( team );

    return 0;
}