30. Structures

Objectives

- Define a `struct`
- Declare `struct` variable with or without initialization
- Use dot **.** operator to access a `struct` variable's member
- Declare and allocation/deallocate memory for a pointer-to-struct variable.
- Use `->` dereferencing operator
- Declare array of `struct` values
- Create nested `struct`.

Up to this point we have been using types provided by C++: basic types, array types, etc.

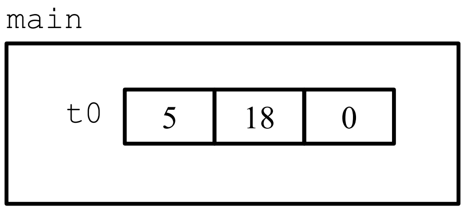
In this set of notes we will create our own types.

`struct`

Try this:

```
#include <iostream>
#include <iomanip>

int main()
{
    struct Time
    {
        int hour;
        int min;
        int sec;
    };

    Time t0;
    t0.hour = 5;
    t0.min = 18;
    t0.sec = 0;

    std::cout << std::setw(2) << std::setfill('0')
              << t0.hour << ':'
              << std::setw(2) << std::setfill('0')
              << t0.min << ':'
              << std::setw(2) << std::setfill('0')
              << t0.sec
              << std::endl;
    return 0;
}
```

`t0` is like a **variable containing other variables**: `hour`, `min`, `and sec`. This is the picture of `main()`'s memory after declaring and assigning values to `t0`:



Here's the format for creating a struct type:

```
struct [struct name]
{
  [type 1] [member name 1];
  [type 2] [member name 2];
  ...
};
```

The most common gotcha is to forget the **semicolon** at the end.

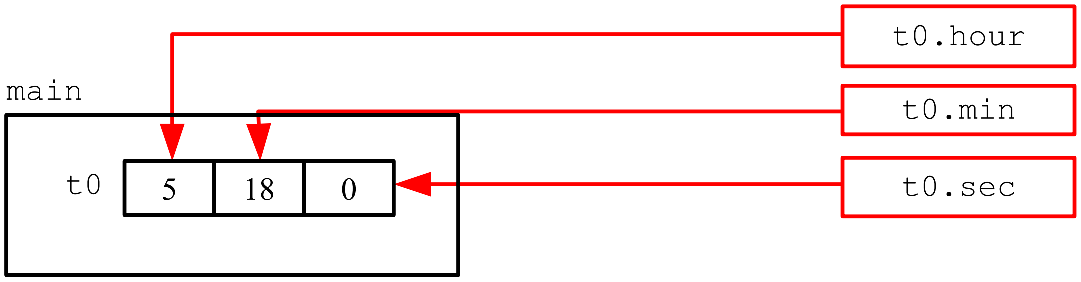
Of course *[type 1]*, *[type 2]*, ... must be types that are already defined.

For our example above, the `hour` in `t0` is called a **member** (or **member variable**) of `t0`.

The member in a struct variable is accessed by the **dot operator**. The format is this:

[struct var name].[member name]

The "`hour` in `t0`" is **`t0.hour`**.



You ***do*** know how to work with *[struct var name].[member name]* since *[struct var name].[member name]* is just like a "regular" variable.

Exercise. Create a variable `t1` of type `Time` and set it to the time one second after `t0`. Print `t1` in the same format as the print statement of `t0`. Note that the values of `sec` of `t1` and `sec` of `t0` are different.

Let's compare the struct variable and an array. Note that a struct variable is like an array in the sense that they both **contain values**.

```
int x[3] = {1, 2, 3};
struct Y
{
    int a;
    int b;
    int c;
};
Y y = {1, 2, 3};
```

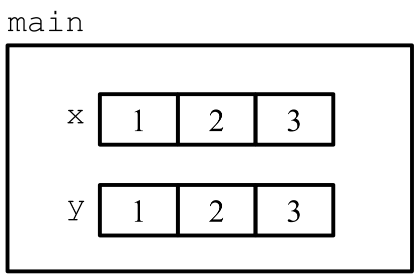
In the above `x` is made up of three integers 1, 2, 3. The variable `y` contains three variables `a`, `b`, and `c` with values 1, 2, and 3 respectively. We refer to the values in `x` by

```
x[0]
x[1]
x[2]
```

while we refer to the values in `y` by

```
y.a
y.b
y.c
```

Make sure you **see the difference!!!** This is the picture of the program's memory during execution:



They both look the same but the names of the cells are different. Remember that.

Note that while you can scan an array like this:

```
for (int i = 0; i < 3; ++i)
{
    std::cout << x[i] << std::endl;
}
```

there's no analog for our `struct` variable `y`: there is no way to scan the values in y using a `for`-loop. That's not the purpose of `struct` variables.

The purpose of `struct` is allow the software engineer to think at a **higher level of abstraction**. For instance in the case of our `Time` struct, a function like

```
void addOneSec(Time & t);
```

and a code like

```
Time currentTime, endTime;

...

while (isLessThan(currentTime, endTime))
{
    // ... do something for a second

    readCurrentTime(currentTime);
}
```

for instance will let us focus on the concept of time and not worry about the details of a time variable (hours, minutes, seconds.) While **functions** chunk up **code**, **struct variables** allows us to chunk up **data**.

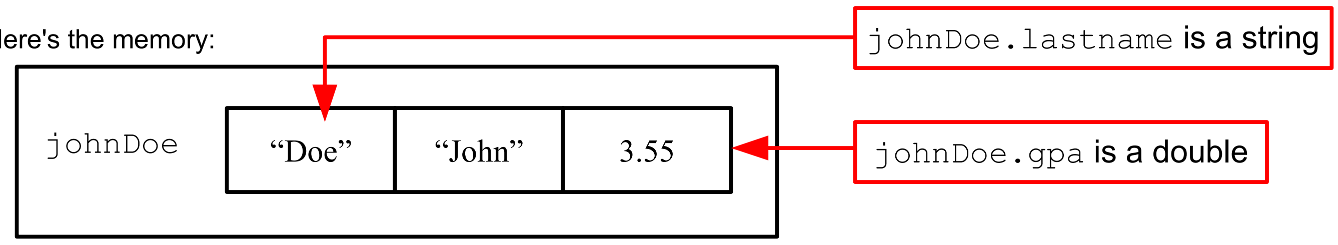
Exercise. There's a section on functions for struct variables. But I'm sure you can handle this exercise. Write the `isLessThan()` function. We need to make sure that the `Time` struct is outside the `main()` and above this function:

```
#include <iostream>
#include <iomanip>

struct Time
{
    int hour;
    int min;
    int sec;
};


bool isLessThan(Time t0, Time t1)
{
  // convert t0 and t1 into seconds since midnight
  // and then compare.
}

int main()
{
    Time t0;
    t0.hour = 5;
    t0.min = 18;
    t0.sec = 0;

    Time t1;
    t1.hour = 5;
    t1.min = 18;
    t1.sec = 1;

    std::cout << isLessThan(t0, t1) << '\n';

    return 0;
}
```

Another difference between arrays and structs is this: You can have member variables of any type in a struct variable. For instance

```
struct Student
{
     char lastname[50];
     char firstname[50];
     double gpa;
};
Student johnDoe = {"Doe", "John", 3.55};
```

Here's the memory:



On the other hand, the values in an array MUST all have the same type. In C/C++ there is no such concept as a single array made up of 3 integers, 4 doubles, and 5 strings.

Exercise. Using this code segment:

```
struct Student
{
     char lastname[50];
     char firstname[50];
     double gpa;
};
Student johnDoe = {"Doe", "John", 3.55};
```

print the first name, last name and GPA of `johnDoe` variable.

Exercise. The following is a `struct` called `Fraction` with two integer members: `numerator` and `denominator`. Correct any errors in the code. Of course a `Fraction` variable models a (mathematical) fraction. Create a `Fraction` variable modeling the fraction 2/3. Write a print function that prints a Fraction variable in the "usual" way, i.e. numerator, followed by '/', and then by the denominator.

```
#include <iostream>

struct Fraction
{
    int numerator;
    int denominator;
};

void print(Fraction f)
{
}

int main()
```

```
{
    Fraction x;
    // assign values to the members of x so that
    // it models the fraction 2/3

    print(x);

    return 0;
}
```

Exercise.

```
#include <iostream>

struct Customer
{
    char fname[100];
    char lname[100];
    char email[100];
};

// Prints customer data in the format
// Firstname: ***, Lastname: ***, Email: ***
void print(Customer customer)
{
}

int main()
{
    Customer customer = {"John",
                         "Doe",
                         "jdoe@gmailcom"};

    print(customer);

    return 0;
}
```

Exercise. The GPS device from a company has an ID (an `int`) and it's position which is longitude (a `double`)and latitude (a `double`). Design a struct that works for the concept of a GPS device. Declare a GPS device with name GPS1 with ID 7, with longitude 45.26 degrees, and with latitude 31.22 minutes. Print the values in GPS1. Prompt the user for a `double x` and add `x` to the longitudinal degree of GPS1. Print the values in GPS1 again.

Exercise. You are writing a game and you need to create game objects. Your game objects have their images loaded from image files. Therefore each game object has a filename associated with it. Your game object must also be drawn to the windows' drawing surface. To do that you need to indicate an area to blit (i.e, copy) the image file. An area on your surface is completely described by the position of the top left corner of the area with respect to the window, i.e. you need the x and y coordinates (as integers). You also need the width and height (as integers) for this area. Each game object also as an energy level which is described by a double. Design a `GameObject` struct that completely describes the concept of a game object outlined above. Declare `redAlien` variable of type `GameObject` whose image is taken from image file “red.bmp”. `redAlien` will be initialized with x = 5, y = 6 and the width and height of the image as it appears on the surface is 20 and 30; it's energy level starts with 100.0.

Exercise. Design a struct called `Car` to be used in a computerized car. What structure members do you need?

One last small point before we're done with this section. Just like in your earlier C++ programs

```
int x;
int y;
```

is the same as

```
int x, y;
```

for a struct. This

```
struct Student
{
    int ID;
    double height;
    double weight;
    double GPA;
};
```

is the same as

```
struct Student
{
    int ID;
    double height, weight, GPA;
};
```

Exercise. Find the error(s):

```
struct X { int x; int y }
X a;
```

By the way it's possible to define a `struct` and use it to declare `struct` variables in a single statement. Make sure you try this:

```
struct Robot
{
    char name[100];
    double weight;
    double height;
    int x;
    int y;
} arnold, t3;
```

And it's even possible to define a `struct` without giving it a name. Run this:

```
struct
{
    char name[100];
    double weight;
    double height;
    int x;
    int y;
} arnold, t3;
```

This is **not common** since in most cases you want to writing functions to accept `struct` values and therefore the function would need to use the `struct`. For instance you might have something like this:

```
#include <iostream>

struct Robot
{
    char name[100];
    double weight;
    double height;
    int x;
    int y;
}

void print(Robot r)
{
    ...
}

int main()
{
    Robot arnold, t3;
    ...

    return 0;
}
```

Initialization

As shown earlier, you can initialize a `struct` variable using initializers:

```
#include <iostream>
#include <iomanip>

int main()
{
    struct Time
    {
        int hour, min, sec;
    };
    Time t0 = {5, 18, 0};

    std::cout << std::setw(2) << std::setfill('0')
              << t0.hour
              << ':'
              << std::setw(2) << std::setfill('0')
              << t0.min
              << ':'
              << std::setw(2) << std::setfill('0')
              << t0.sec
              << std::endl;

    return 0;
}
```

Notice that this looks very similar to initialization of an array. And just like arrays you can perform partial initialization:

```
struct Time
{
    int hour, min, sec;
};
struct Time t0 = {5};
```

Exercise. What are the values of `t0`?

```
struct Time
{
    int hour, min, sec;
};
Time t0 = {5};
```

Exercise. What are the values in x?

```
struct X
{
    int a, b, c;
    double d, e, f;
};
X x = {5};
```

Exercise. What is the output?

```
struct X
{
    int x;
    char y;
    double z;
};

X a = {1, '$', 3.14};
std::cout << a[0] << ' ' << a[1] << ' ' << a[2]
          << std::endl;
```

Exercise. Write one single input statement (of course using `std::cin`) to fill `a` with values and one statement to print all the values in `a`.

```
struct X
{
    int x;
    char y;
    double z;
};
X a;
// input values into a
// output all values in a
```

Exercise. It's not too surprising that you can put an array into a struct. Define a `TicTacToeGame` struct containing `board`, a 3-by-3 array of characters and a `turn` variable that is of `char` type. The array `board` models the state of a tic-tac-toe game and the `turn` variable tells us which player (either X or O) will be the next to make a move.

```
// define struct here


// declare variable x of type TicTacToe so that it
// models the game board
//  X |   | O
// ---+---+---
//  X | X | O
// ---+---+---
//    |   |
// (for instance x.board[0][0] is 'X' and the turn is
// set to 'O'.


// Write a double for-loop to print x.board in the
// format of the above comment. Of course the code
// should work for different x.board values!!!


// Now write a statement to prompt the appropriate
// player to make a move. For instance if x.turn
// is 'X', the message printed is "X's turn: " and
// if x.turn is 'O', then the message is "O's turn: "
```

Operator =

To copy values from one struct variable to another you can of course do this:

```
struct Time
{
  int hour, min, sec;
};
Time t0 = {5, 18, 0};
Time t1;

t1.hour = t0.hour;
t1.min = t0.min;
t1.sec = t1.sec;
```

A better way is this:

```
struct Time
{
  int hour, min, sec;
};
Time t0 = {5, 18, 0};
Time t1;

t1 = t0;
```

In other words the operator = (i.e. assignment operator) is defined between struct variables of the same type. It copies all the values from the left to the right. It's not too surprising that the same holds for the initialization operator:

```
struct Time
{
    int hour, min, sec;
};
Time t0 = {5, 18, 0};
Time t1 = t0;
```

Exercise. Is the operator `==` defined for struct variables of the same type? In other words does the following print 1?

```
struct Time
{
    int hour, min, sec;
};
Time t0 = {5, 18, 0};
Time t1 = t0;

std::cout << (t1 == t0) << std::endl;
```

Exercise. Is the operator `!=` defined for struct variables of the same type? Design an experiment to verify your guess.

Array of Structs

Does this surprise you?

```
#include <iostream>
#include <iomanip>


struct Time
{
    int hour;
    int min;
    int sec;
};


int main()
{
    Time t[10];

    for (int i = 0; i < 10; i++)
    {
        t[i].hour = 5;
        t[i].min = 18;
        t[i].sec = 2 * i;
    }


    for (int i = 0; i < 10; i++)
    {
        std::cout << std::setw(2)
                  << std::setfill('0')
                  << t[i].hour
                  << ':'
                  << std::setw(2)
                  << std::setfill('0')
                  << t[i].min
                  << ':'
                  << std::setw(2)
                  << std::setfill('0')
                  << t[i].sec
                  << std::endl;
    }

    return 0;
}
```

Enough said ...

Pointers to structs

Since you can create pointers to any kind of variable, you can create pointers to structs too.(Review your notes/books on pointers if necessary.) Here's a quick review.

Recall that to declare a pointer to point to a value of type X you do this:

```
      X * p;
```

For instance here's an int pointer:

```
      int * p;
```

If you are pointing to a value that already exists you can access that value using your pointer:

```
int x = 5;
int * p = &x;
*p = 42;
std::cout << p << std::endl
std::cout << (*p) << std::endl
```

If not you have to give your pointer a value to point to. This is how you do it:

```
int * p;
p = new int;
```

That's called **allocating memory** for `p`. (You have already seen this concept.) You can do it in one step:

```
int * p = new int;
```

Where does this memory (the int) come from? It's from a pool of available memory that was created before your program runs; it's called the heap or the free store. Once you're done with using the value your pointer is pointing to you must release the memory back to the heap. This is called **deallocating memory**. You do this:

```
int * p = new int;
... do something with p ...
delete p;
```

Note (yet again) that `delete p` releases the memory that `p` is pointing to; `p` itself is not somehow removed from the program until it goes out of scope. Therefore you can use it again:

```
int * p = new int;
... do something with p ...
delete p;
...
p = new int;
... do something else with p ...
```

```
delete p;
...
p = new int;
... do something else with p ...
delete p;
```

You can also get pointers to point to an array of values in the heap:

```
int * p = new int;
... do something with p ...
delete p;
...
p = new int;
... do something else with p ...
delete p;
...
int size = 42;
p = new int[size];
... do something with p ...
delete [] p;
```

Once p points to an array, you can use p like an array:

```
int size = 42;
p = new int[size];
for (int i = 0; i < size; i++)
{
    p[i] = i * i;
}
delete [] p; // <--- WARNING!!!
```

The **address** of the i-th value that p points to is `p + i` and therefore the value of this value is `*(p+i)`. So remember that the i-th value of the array that p points is is both `p[i]` and `*(p + i)`. They are the same. The preference is to use `p[i]`.

This is only a quick review. You should go over your previous notes on pointers. Now try this:

```
#include <iostream>
#include <iomanip>

int main()
{
  struct Time
  {
    int hour;
    int min;
    int sec;
  };

  Time t0;
  t0.hour = 5;
  t0.min = 18;
  t0.sec = 0;

  std::cout
    << std::setw(2) << std::setfill('0') << t0.hour
    << ':'
    << std::setw(2) << std::setfill('0') << t0.min
    << ':'
    << std::setw(2) << std::setfill('0') << t0.sec
    << std::endl;

  Time * p = new Time;
  *p = t0;

  std::cout
    << std::setw(2) << std::setfill('0') << (*p).hour
    << ':'
    << std::setw(2) << std::setfill('0') << (*p).min
    << ':'
    << std::setw(2) << std::setfill('0') << (*p).sec
    << std::endl;

  delete p;

  return 0;
}
```

Read the code carefully. There is really no new concepts at all. While `p` is the pointer to a `Time` value, `*p` is the actual `Time` value `p` is pointing to. Therefore

```
(*p).hour
```

is the hour of the value p is pointing to. There is actually a shorthand notation for this. Note that `(*p)`.hour involves two operators: the * and the . operators. Let me emphasize that:

`(*p).hour`

is the same as

`p->hour`

The operator -> is called the **de-referencing operator**. Unfortunately * as in `*p` is also called the de-referencing operator!!! You should use this instead of doing de-referencing-followed-by-the-dot-operator, i.e. write

```
p->hour          GOOD
(*p).hour        BAD
```

Of course one quick question to ask is this. Suppose we omit the parentheses so that we have two operators, like this `*x.y`. Which operator goes first? Is `*x.y` really `(*x).y` or is it `*(x.y)`? The dot operator actually goes first. In other words `*x.y` is really `*(x.y)`.

Exercise. Rewrite the above program with the -> operator.

Now try this:

```
#include <iostream>
#include <iomanip>

int main()
{
  struct Time
  {
    int hour;
    int min;
    int sec;
  };

  Time t0;
  t0.hour = 5;
  t0.min = 18;
  t0.sec = 0;

  std::cout
    << std::setw(2) << std::setfill('0') << t0.hour
    << ':'
    << std::setw(2) << std::setfill('0') << t0.min
    << ':'
    << std::setw(2) << std::setfill('0') << t0.sec
    << std::endl;

  Time * p = new Time[100];

  for (int i = 0; i < 100; i++)
  {
    p[i] = t0;
    p[i].sec++;
    std::cout
    << std::setw(2) << std::setfill('0') << p[i].hour
    << ':'
    << std::setw(2) << std::setfill('0') << p[i].min
    << ':'
    << std::setw(2) << std::setfill('0') << p[i].sec
    << std::endl;
  }
  delete [] p;

  return 0;
}
```

Study the code carefully. Again there is really no new concept.

Exercise. Refer to a previous exercise on the `Fraction` struct:

```
#include <iostream>

struct Fraction
{
    int numerator;
    int denominator;
}

void print(Fraction f)
{
}

int main()
{
    Fraction x;
    // assign values to the members of x so that
    // it models the fraction 2/3

    print(x);

    return 0;
}
```

Modify this program and prompt the user for `size` (an integer). Declare a pointer `Fraction` and allocate an array of `Fraction`s to this pointer; the size of this array is given by `size`. In a for-loop, put random fractions into the array (there is one value you should avoid in the denominators ...) Print all the fractions. Deallocate memory used by the pointer.

Functions

Look at this again:

```
#include <iostream>
#include <iomanip>

int main()
{
  struct Time
  {
    int hour;
    int min;
    int sec;
  };

  Time t0;
  t0.hour = 5;
  t0.min = 18;
  t0.sec = 0;

  std::cout
    << std::setw(2) << std::setfill('0') << t0.hour
    << ':'
    << std::setw(2) << std::setfill('0') << t0.min
    << ':'
    << std::setw(2) << std::setfill('0') << t0.sec
    << std::endl;

  return 0;
}
```

Exercise. Write a function to print the values of a `Time` variable:

```
#include <iostream>
#include <iomanip>

void print(Time t)
{
}

int main()
{
  struct Time
  {
    int hour;
    int min;
    int sec;
  };

  Time t0;
  t0.hour = 5;
  t0.min = 18;
  t0.sec = 0;

  print(t0);

  return 0;
}
```

(Where must the struct definition be?)

Once you use a function for passing variables of a certain type you immediately ask if the parameters (the variables in the function) can modify the values in the calling function ... try this:

```
#include <iostream>
#include <iomanip>

// Time struct here

void print(Time t)
{
}

void init(Time t)
{
  t.hour = t.min = t.sec = 0;
}

int main()
{

  Time t0;
  t0.hour = 5;
  t0.min = 18;
  t0.sec = 0;

  print(t0);
  init(t0);
  print(t0);

  return 0;
}
```

By default, structs are not passed by reference. Recall that functions can modify an array which is passed in. Therefore arrays are passed by reference by default. This is another difference between an array and a struct variable.

To really change the variable in the calling function you can do two things (see your earlier notes): using references or using pointers. Here's the version that uses references
:

```
...
```

```
void init(Time & t)
{
  t.hour = t.min = t.sec = 0;
}

int main()
{
  Time t0;
  t0.hour = 5;
  t0.min = 18;
  t0.sec = 0;

  print(t0);
  init(t0);
  print(t0);

  return 0;
}
```

And here's an initialization using pointers::

```
...

void init2(Time * t)
{
  t->hour = t->min = t->sec = 0;
}

int main()
{
  ...

  print(t0);
  init2(&t0);
  print(t0);

  return 0;
}
```

Pass by Constant Reference

You know that since `struct` variables are passed by value, they can't be changed by function call unless if passed by reference. Now look at your print function:

```
#include <iostream>
#include <iomanip>

// Time structure here

void print(Time t)
{
}

int main()
{
  ...
  return 0;
}
```

This does work since the `print()` function should not change the value of the variable passed in anyway. However a struct variable by nature tends to contains lots of values. If you pass by value, the values will be copied to the receiving parameter. This takes time. Therefore you can pass by reference to save time. Pass by reference will not result in values being copied. Therefore we can do this:

```
#include <iostream>
#include <iomanip>

// Time structure here

void print(Time & t)
{
}

int main()
{
  ...
  return 0;
}
```

Now the danger is that you (or someone working with you on the same project) might accidentally change the value of `t` in the `print()` function:

```
#include <iostream>
#include <iomanip>

// Time structure here

void print(Time & t)
{
   t.hour = 0; // YIKES!!!
}

...
```

So one way to prevent that and at the same time passing a `Time` variable by reference for speed is to make the parameter a **constant reference**:

```
#include <iostream>
#include <iomanip>

// Time structure here

void print(const Time & t)
{
}

...
```

In summary

- **Struct parameters** should be **pass by reference**
- If a function **should not change** the values of a **struct parameter**, then the parameter should be a **constant reference**.

Exercise. Recall the following struct:

```
struct Fraction
{
    int numerator;
    int denominator;
}
```

Write a `print()` function for this struct. Make sure you use pass-by-constant-reference.

Exercise. Write a function that checks if a `Time` struct parameter is valid. The boolean value `true` is returned if the value is correct, i.e. the hour is between 0 and 23 (inclusive), the minute is between 0 and 59 (inclusive), and the second is between 0 and 23 (inclusive). The name of the function is `isValid`. [Question: Does the function change the value of the parameter? What type of parameter-passing should you use?]

Struct return values

You already know that functions can only return one value at one time. So something like this is **WRONG**:

```
int, int squareAndCube(int x)
{
  return x*x, x*x*x; // trying to return a square
                     // and a cube ...
}
```

In other words you cannot return two values at the same time.

However all the values of the member variables inside a struct variable is considered a single value. So this is OK:

```
#include <iostream>
#include <iomanip>

// Time struct

// print function

Time addOneSec(const struct Time & t0)
{
  struct Time t = t0;
  t.sec++;
  if (t.sec >= 60)
  {
    t.min += t.sec / 60;
    t.sec %= 60;
    if (t.min >= 60)
    {
      t.hour += t.min / 60;
      t.min %= 60;
      if (t.hour >= 24)
      {
        t.hour %= 24;
      }
    }
  }
  return t;
}


int main()
{
  struct Time t0 = {5, 18, 0}; // 05:18:00 hours

  ...

  t0 = addOneSec(t0);
  print(t0);

  return 0;
}
```

Header Files

A `struct` that is used by different programs can of course be kept in a header file. Function prototypes for functions related to the `struct` can be placed in the header file. And of course the implementation of the functions are in a cpp file.

Exercise. Rewrite this:

```
#include <iostream>
#include <iomanip>

// Time struct

// print function

Time addOneSec(const struct Time & t0)
{
  struct Time t = t0;
  t.sec++;
  if (t.sec >= 60)
  {
    t.sec %= 60;
    t.min += t.sec / 60;
    if (t.min >= 60)
    {
      t.min %= 60;
      t.hour += t.min / 60;
      if (t.hour >= 24)
      {
        t.hour %= 24;
      }
    }
  }
  return t;
}


int main()
{
  struct Time t0 = {5, 18, 0}; // 05:18:00 hours

  ...

  t0 = addOneSec(t0);
  print(t0);

  return 0;
}
```

So that it looks like this:

```
#include <iostream>
#include <iomanip>
```

```
#include "Time.h"


int main()
{
  struct Time t0 = {5, 18, 0}; // 05:18:00 hours

  ...

  t0 = addOneSec(t0);
  print(t0);

  return 0;
}
```

Of course you need a `Time.h` and a `Time.cpp`.

Nested `struct`

A `struct` definition contains variable of any type. Since a `struct` itself is a type ... **you can put a `struct` in a `struct`!!!!**

Try this:

```
struct StopWatch
{
      int ID;
      Time time;
};

StopWatch s;
s.ID = 0;
s.time.hour = 5;
s.time.min = 18;
s.time.sec = 0;
```

Nothing to it right?

Exercise. Write a `Date` struct containing month, day, year integers and a `Time` struct containing hour, minute, second integers. Finally create a `DateTime` struct that contains the date of `Date` type and time of the `Time` type. Write a `print()` function for `Date`, `Time`, and `DateTime` types. The `print()` function for `DateTime` should be the `print()` from `Date` and `Time`.

Exercise. Given

```
struct X
{
    int x;
    int * y;
    int z;
};

struct Y
{
    int * x;
    X * y;
    int * z;
};

struct Z
{
    X * x;
    Y * y;
    int * z;
};
```

and `x`, `y`, `z`, `w` are variables of type `X`, `Y`, `Z`, `Z*` respectively. Which of the following is valid:

- `std::cout << x.x + 2;`
- `x.y = new int;`
- `x.z = new int[10];`
- `delete [] y.z;`
- `y.x = x.y;`
- `std::cout << y->y.x;`
- `std::cout << y.y->x;`
- `std::cout << z.y->x;`
- `std::cout << z->y->x;`
- `std::cout << z.y->y->x;`
- `std::cout << w->y->y.x;`
- `std::cout << w->y->y->x;`

Note that if there are several `->` operators in a chain, then the evaluation goes left-to-right. In other words:

```
a->b->c->d
```

is the same as

```
((a->b)->c)->d
```

And remember that `.` goes before `->`. So

```
a->b->c.d
```

is really the same as

```
a->b->(c.d)
```