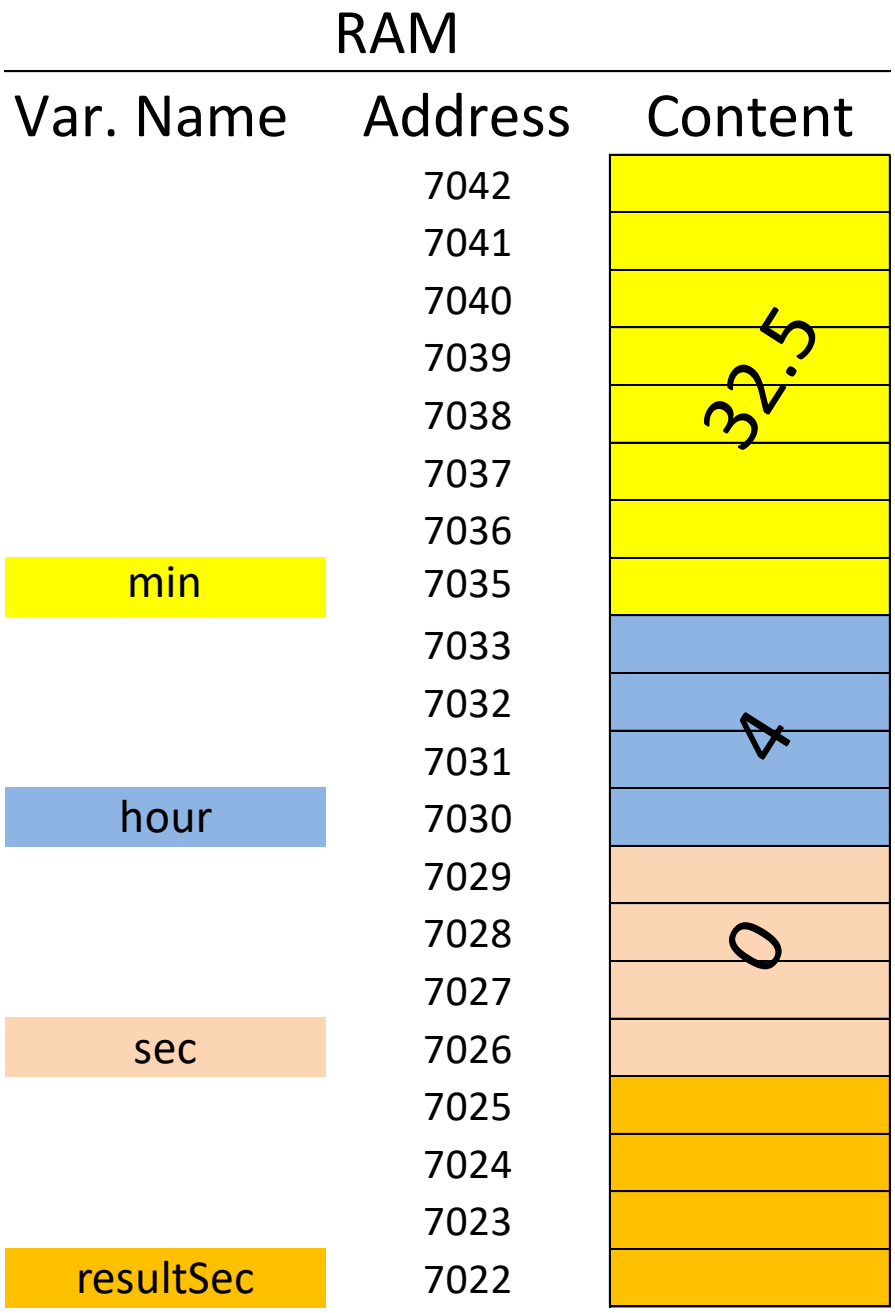
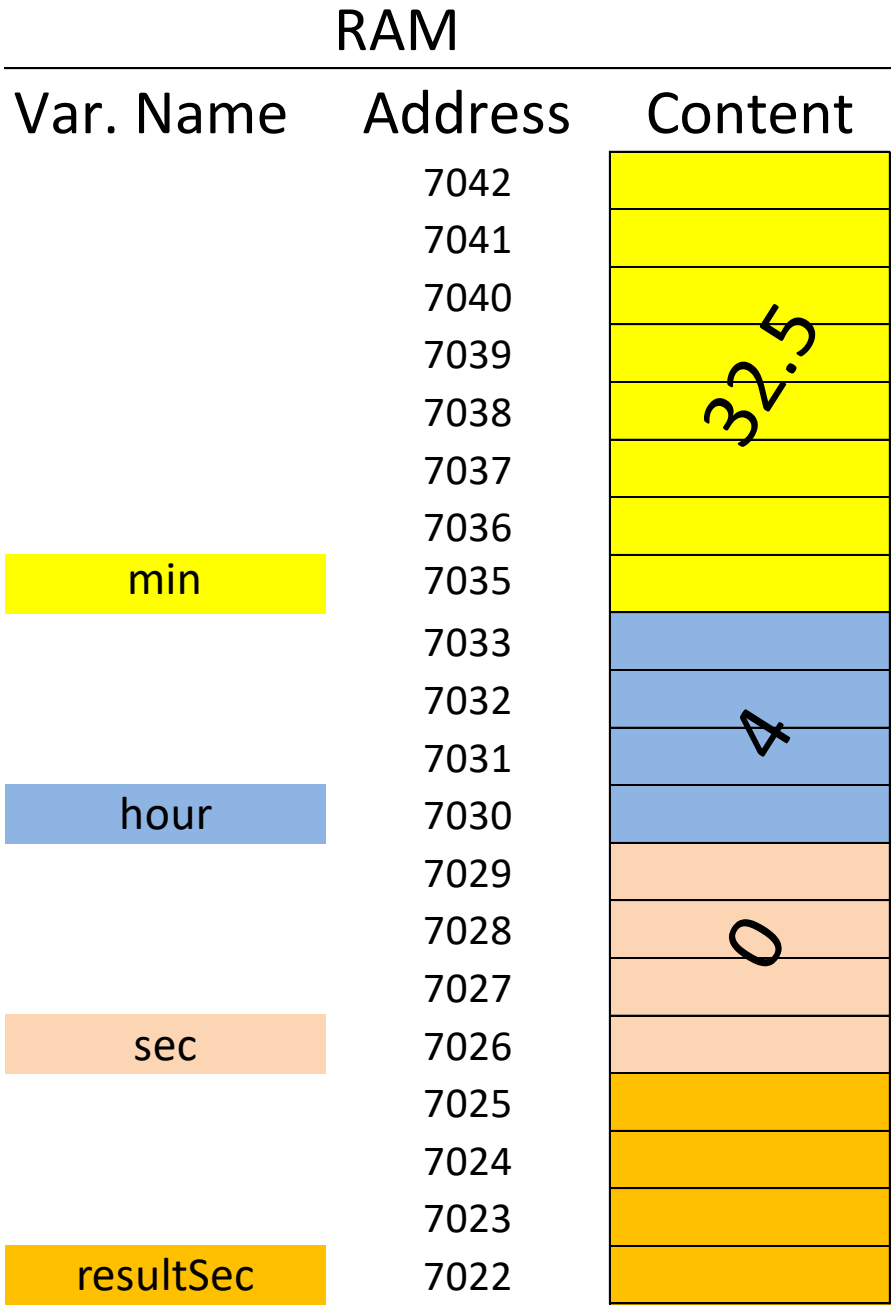
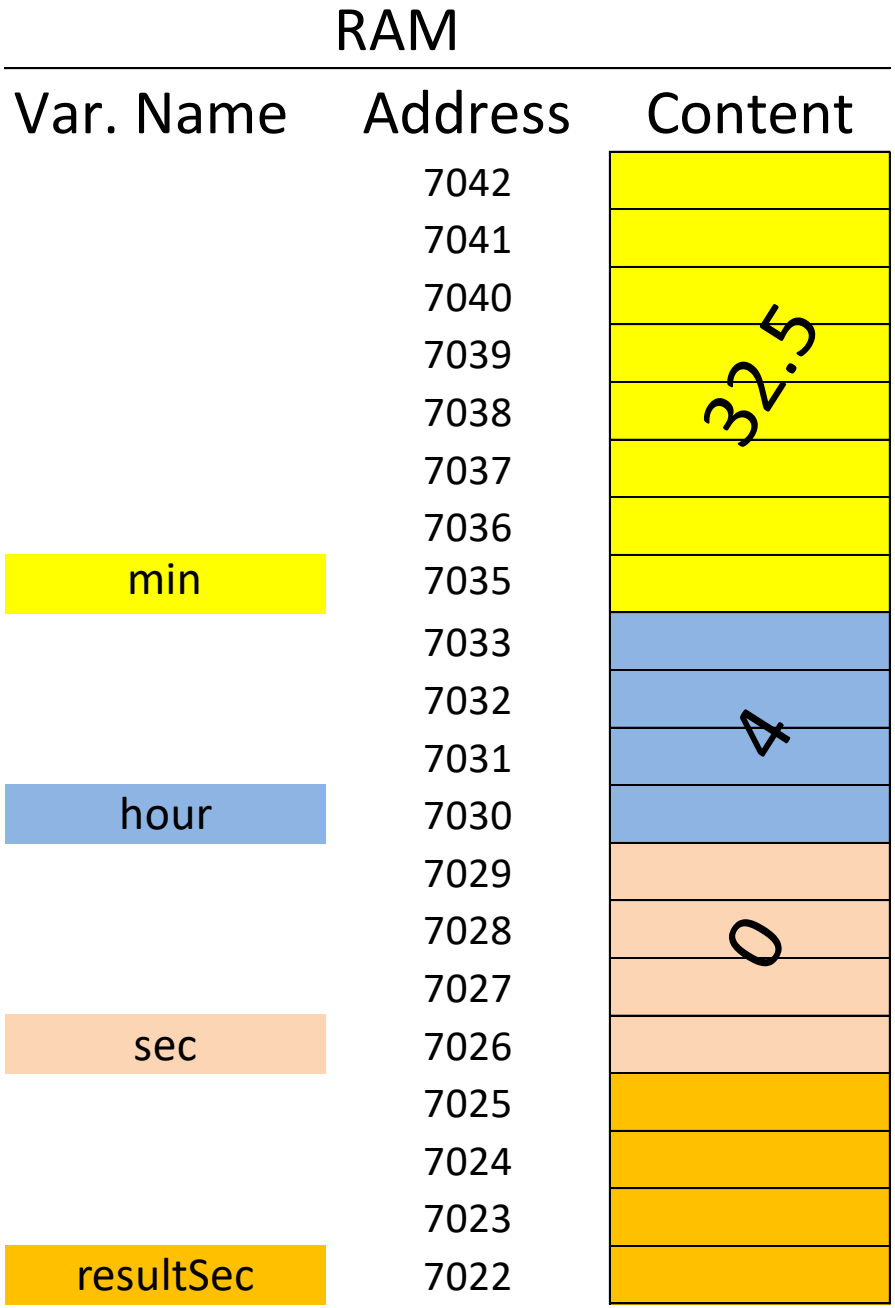


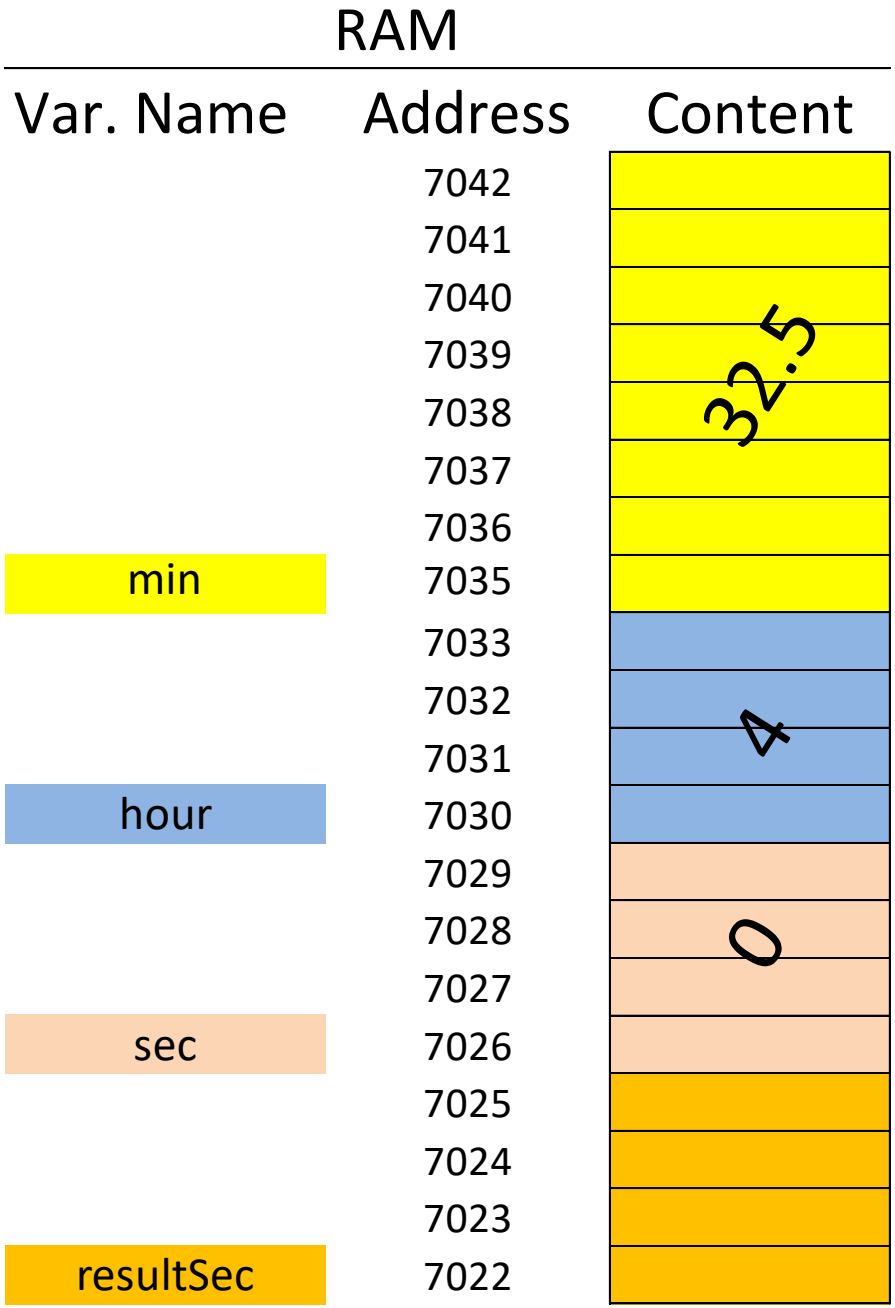
Thus far, when we have used the ***name*** of a variable, the compiler has known that we mean to use the ***value*** of that variable, i.e., the ***content*** of the RAM



So, if we call a function and pass a variable to the function for its use, it's the ***value*** that's passed – i.e. makeSeconds(***min***) would result in passing ***32.5*** to makeseconds()



The & operator applied to a variable means to <u>not</u> use the value, but rather to use the ***ADDRESS*** of the variable, so ***&min*** is equivalent to 7035, not 32.5



If makeSeconds() is declared as having a ***double&*** argument, then makeSeconds(min) passes the ***address*** 7035 to the function, rather than the value 32.5

- “Call by value” is what we have thus far done, passing in the ***value*** of a variable. That value is available as a ***local*** variable within the function.

- “Call by reference” is the other method, in which we pass the ***address*** of the variable, which the function can then use in order to access/modify the ***non-local*** (original) variable storage location.

Call by value:

```
int makeSeconds(double min)  {
        cout << min;    // This writes 32.5
        min = 40.0;     //  This changes the local value
                        //   but not the global
        cout << min;    //  This writes 40.0
}

double min = 32.5;
main() {
        makeSeconds(min);
        cout << min;     // This writes 32.5
}
```

Call by reference:

```
int makeSeconds(double&  min)  {
        cout << min;    // This writes 32.5
        min = 40.0;     //  This changes the global value
        cout << min;    //  This writes 40.0
}

double min = 32.5;
main() {
        makeSeconds(min);
        cout << min;     // This writes 40.0
}
```