# CS 31 Discussion 1A, Week 8

Zengwen Yuan (zyuan [at] cs.ucla.edu)

Humanities A65, Friday 10:00—11:50 a.m.

# Today's focus

- Pointer
- Structure
- Clarifications

# Pointer

- A pointer is the memory address of a variable.
- It provides direct access to manipulate values in memory
- Pointer v.s. Variable:
    - Pointer holds memory address
    - Variable holds some value

# Pointer: love-hate relationship

MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A 0x6339392C, 0x7363682E.
I HATE YOU.

http://imgs.xkcd.com/comics/pointers.png

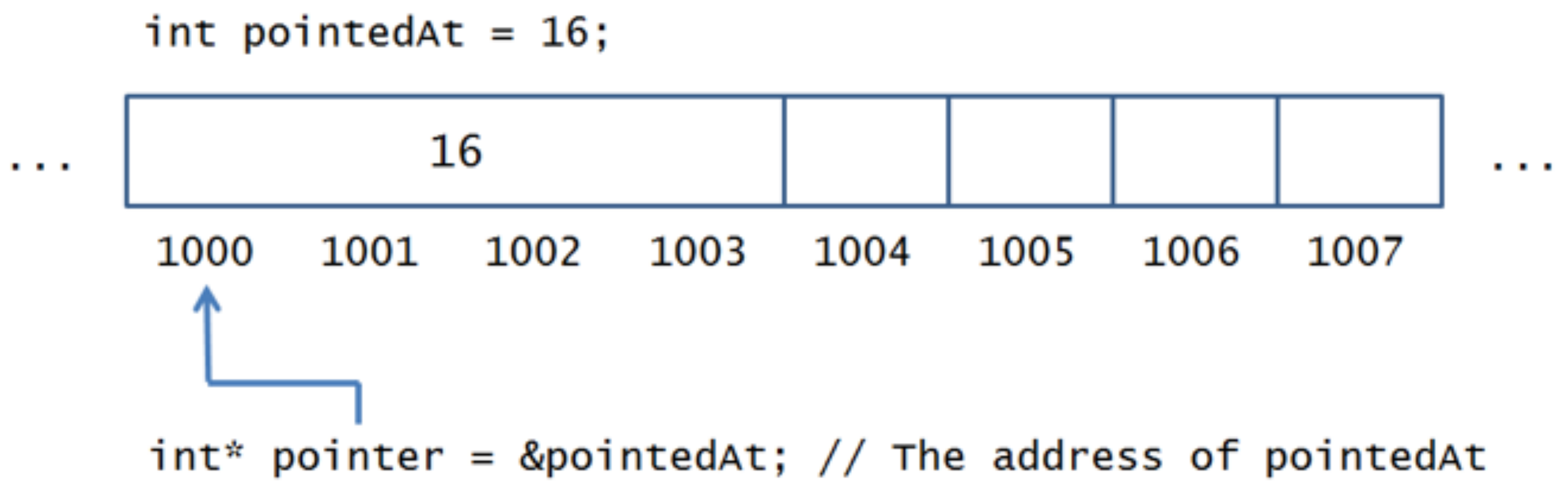
# Pointer: basics

- Pointer variable (don’t confuse with the variables!)

```
<type> *<name>;
```

```
int    *ip;    // pointer to an integer
double *dp;    // pointer to a double
float  *fp;    // pointer to a float
char   *ch     // pointer to character
```

- Pointer representation in memory



```
int pointedAt = 16;
```

```
int* pointer = &pointedAt; // The address of pointedAt
```

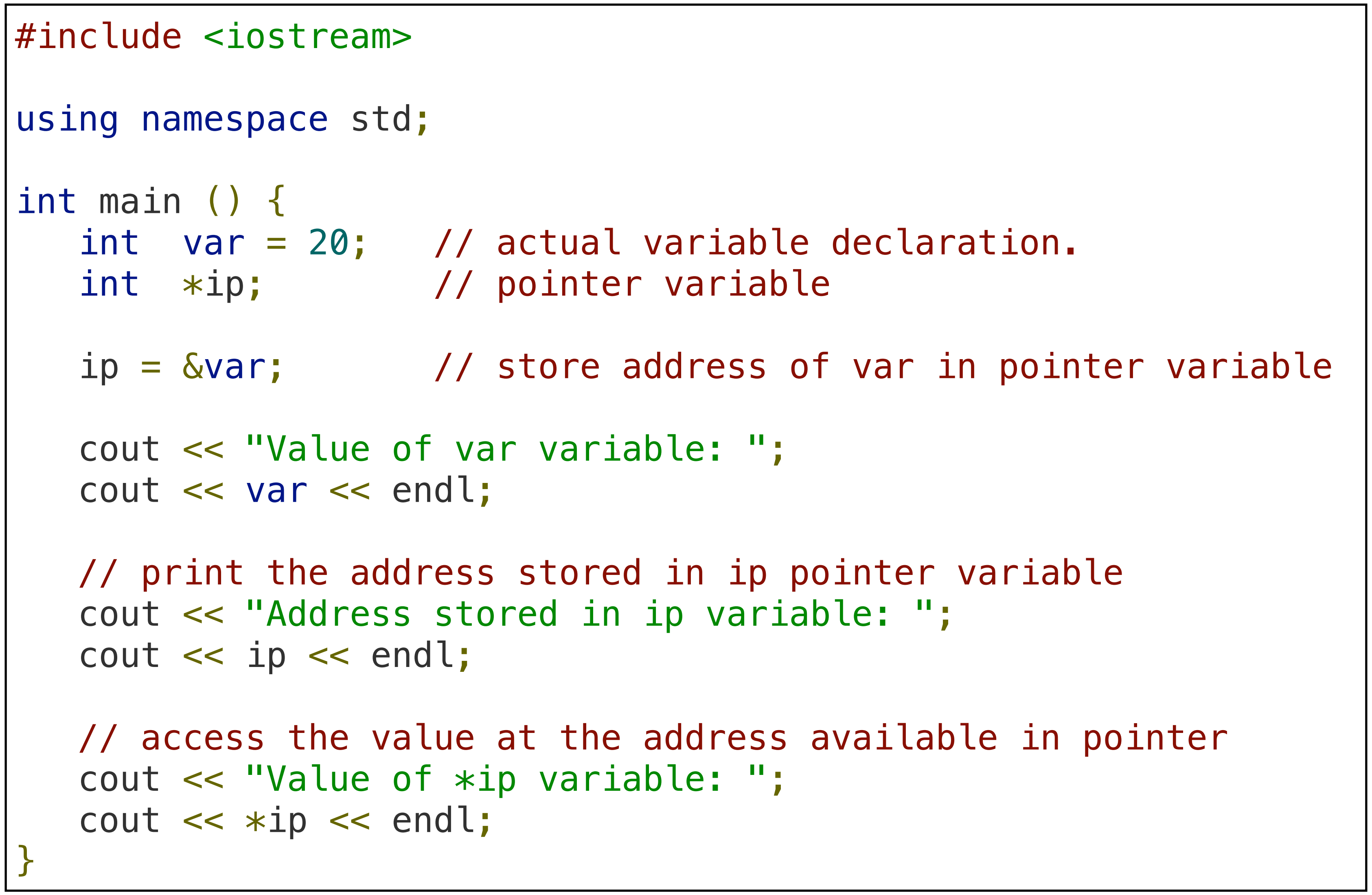
*courtesy: Andrew Forney*

# Pointer: important operations

- Most common patterns when you use pointer
  - (a) define pointer variables
  - (b) assign the address of a variable to a pointer
  - (c) finally access the value at the address available in the pointer variable.

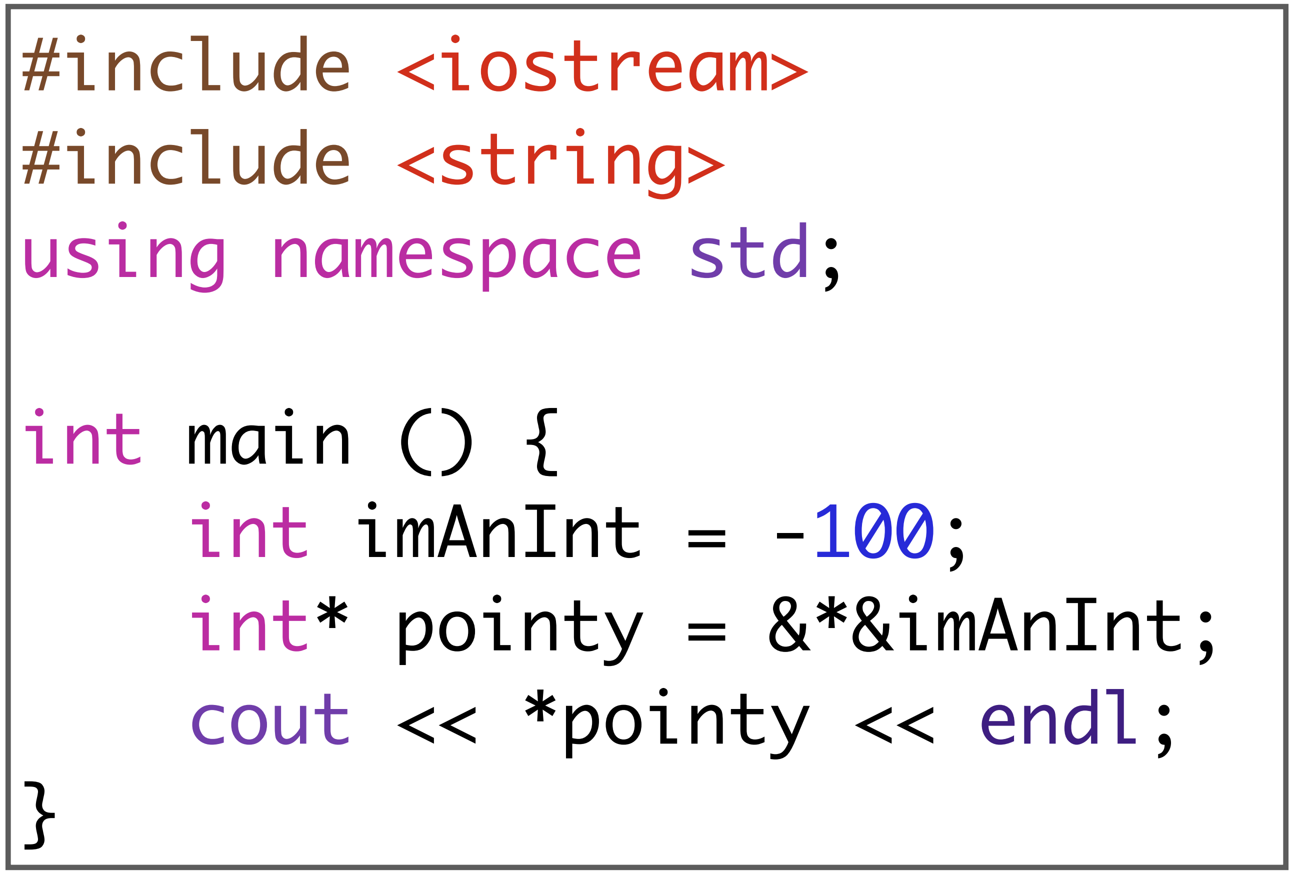
# Pointer: access variable values

- The & and * operators — get address and dereference



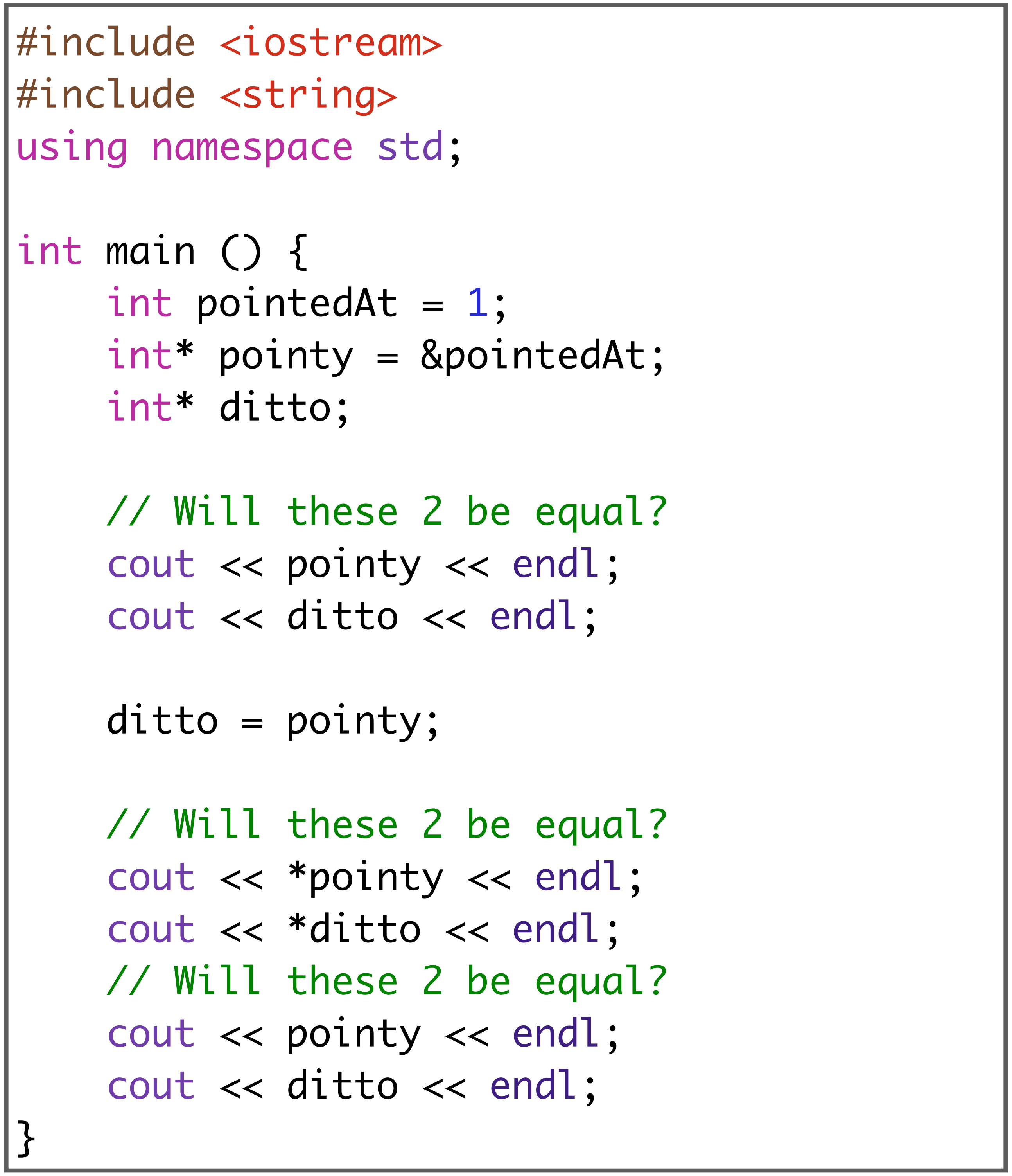
```cpp
#include <iostream>

using namespace std;

int main () {
   int  var = 20;   // actual variable declaration.
   int  *ip;        // pointer variable

   ip = &var;       // store address of var in pointer variable

   cout << "Value of var variable: ";
   cout << var << endl;

   // print the address stored in ip pointer variable
   cout << "Address stored in ip variable: ";
   cout << ip << endl;

   // access the value at the address available in pointer
   cout << "Value of *ip variable: ";
   cout << *ip << endl;
}
```

# Pointer: example



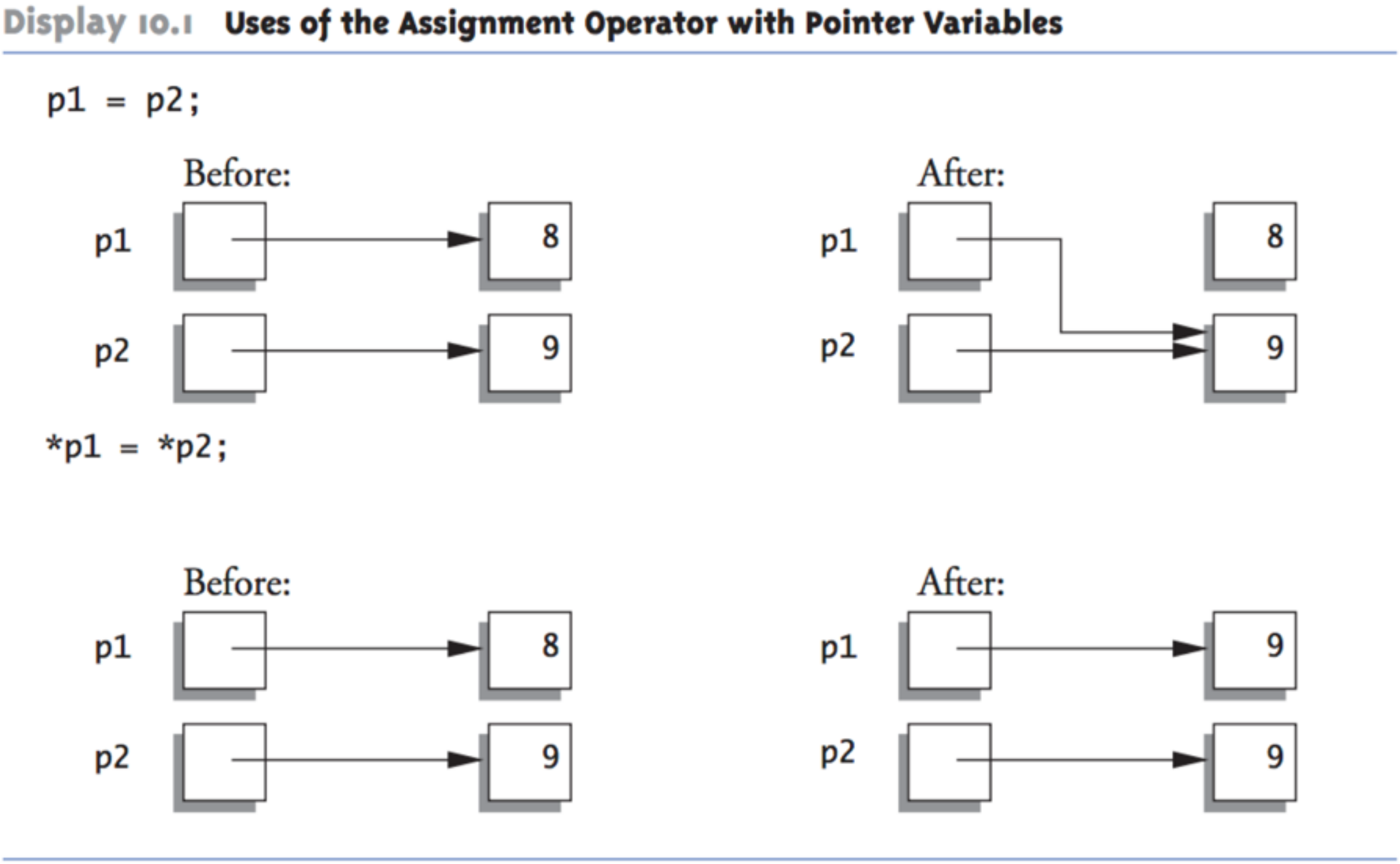
```
#include <iostream>
#include <string>
using namespace std;

int main () {
    int imAnInt = -100;
    int* pointy = &*&imAnInt;
    cout << *pointy << endl;
}
```

# Pointer: example



```
#include <iostream>
#include <string>
using namespace std;

int main () {
    int pointedAt = 1;
    int* pointy = &pointedAt;
    int* ditto;

    // Will these 2 be equal?
    cout << pointy << endl;
    cout << ditto << endl;

    ditto = pointy;

    // Will these 2 be equal?
    cout << *pointy << endl;
    cout << *ditto << endl;
    // Will these 2 be equal?
    cout << pointy << endl;
    cout << ditto << endl;
}
```

# Pointer: assignment operator



**Display 10.1 Uses of the Assignment Operator with Pointer Variables**

```
p1 = p2;
```

```
*p1 = *p2;
```

# Pointer: null pointer

- Sometimes we return a null pointer (e.g. to indicate an error)

- Two flavors:

  - `NULL` — a constant, 0.

  - `nullptr` — introduced in C++11. A literal constant.

# Pointer: null pointer example

```
double* findFirstNegative(double a[], int n)
{
    for (double* p = a; p < a + n; p++)
    {
        if (*p < 0)
            return p;
    }
    return nullptr;
}
...
double* p = findFirstNegative(da, 5);
if (p == nullptr)
...
else
...
```

# Pointer: null pointer example

```cpp
#include <iostream>
#include <string>
using namespace std;

int main () {
    int i = 50;
    int* latePointer = nullptr;

    if (latePointer == nullptr) {
        latePointer = &i;
    } else {
        cout << "<_< >_>" << endl;
    }
    cout << *latePointer << endl;
}
```

# *Pointer: dynamic variables/arrays

- Duh… I am not expected to teach you these this week!

- new operator

dynamic variable

```
MyType *p;
p = new MyType;
```

- delete operator

# Pointer: use pointers with arrays

- The array name can be used as a pointer variable:
  - Traversing array

```
const int MAXSIZE = 5;
double da[MAXSIZE];
double* p;
...
for (double* p = da; p < da + MAXSIZE; p++)
*p = 3.6;
```

  - Passing arrays or portions to a function

```
int findFirstNegative(const double a[], int n);
// or its equivalent
int findFirstNegative(const double* a, int n);
...
double b[5];
...
cout << findFirstNegative(b, 5);
cout << findFirstNegative(b+2, 3);
```

# Pointer: pointers with arrays example

```
#include <iostream>
#include <string>
using namespace std;

int main () {
    int i[][3] = {
        {1, 2, 3},
        {4, 5, 6}};

    int*  pointy = &i[1][1];
    int*  copyPointy = pointy;
    *pointy = 100;
    pointy = &i[0][2];

    cout << *pointy << endl;
    cout << *copyPointy << endl;
}
```

# Structure

- Structure is a collection of values of different types
  - i.e. student record (int UID, string name) …
- Variables inside a structure are member variables
- An instance of a structure is an object
  - i.e. a student record for Bob is (104000000, “Bob”)

```
struct <structName> {
    <member1_type> <member1_name>;
    <member2_type> <member2_name>;
    // ...etc.
}; // Remember the semicolon!
```

# Structure: declaration and the dot operator

- Use dot operator to specify a member variable of a structure variable

```
struct Employee
{
    string name;
    double salary;
    int age;
}; // DON'T FORGET THE SEMICOLON!!!

// Dot operator
Employee e1;
Employee e2;
e1.name = "Fred";
e1.age = 60;
e2.name = “Ethel";

// Use struct array
Employee company[100];
company[1].name = “Ricky";
// ...
```

# Structure: use pointer and the arrow operator

- When using a pointer, you can use -> to specify a member variable (**`p->m`** means the same as **`(*p).m`**)

```
struct Employee
{
    string name;
    double salary;
    int age;
}; // DON'T FORGET THE SEMICOLON!!!

// use the array operator -> to set values
// this is more convenient and preferred
Employee *ep1;
ep1 = new Employee;
ep1->name = "Fred";
ep1->age = 60;

// equivalently, you can use pointer
// dereference and the dot operator
// to set values
Employee *ep2;
ep2 = new Employee;
(*ep2).name = "Ethel";
```

# Structure: constructor

- A constructor is used to initialize the data members of a newly declared struct object

```
#include <iostream>
#include <string>
using namespace std;

struct Ford {
    // Data Member
    int tires;
    string model;
    // Constructor for Ford objects
    Ford () {
        tires = 5; // 4+1 Spare
        model = "Ranger";
    }
};

int main () {
    // Constructor invoked below!
    Ford myCar;

    cout << myCar.tires << endl;
    cout << myCar.model << endl;
}
```

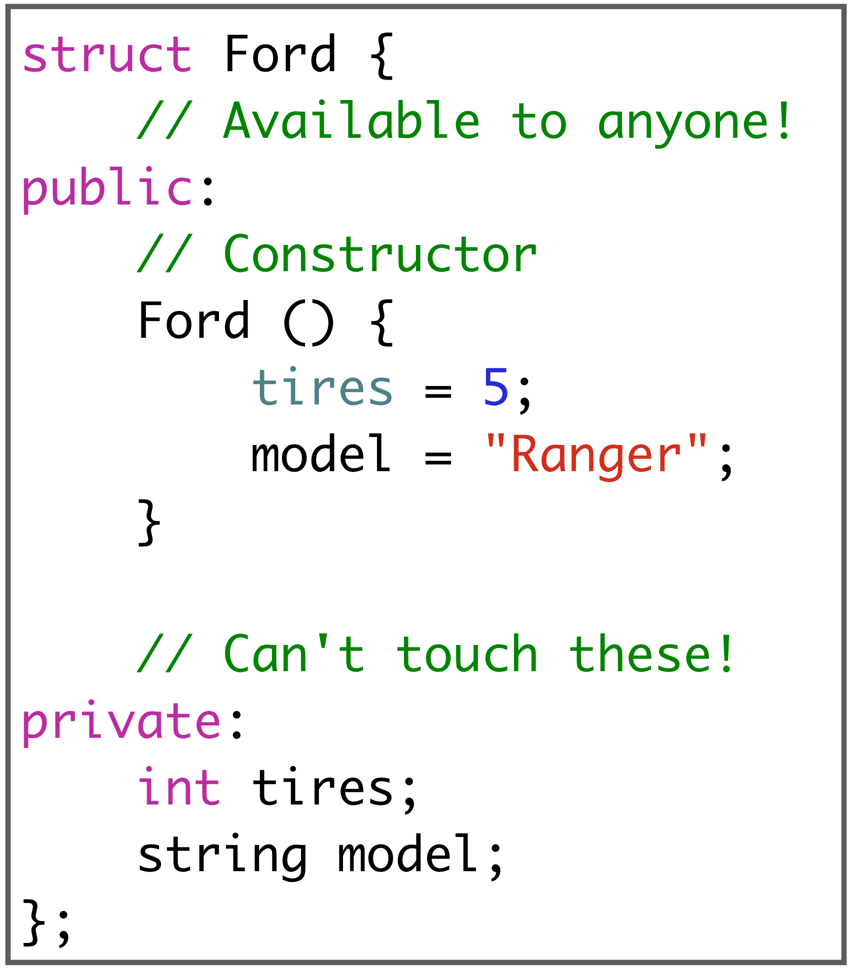
# Structure: public & private tags

- Motivation: ensure the integrity of an object's data by restricting who can modify them

- The public tag in a struct, says "Everything that comes after this (until you say otherwise) is publicly accessible and modifiable."

- The private tag in a struct, says "Everything that comes after this (until you say otherwise) is ONLY accessible to member functions."

# Structure: public & private tags example



```
struct Ford {
    // Available to anyone!
public:
    // Constructor
    Ford () {
        tires = 5;
        model = "Ranger";
    }

    // Can't touch these!
private:
    int tires;
    string model;
};
```

- But how would you be able to access private members?

# Structure: member function

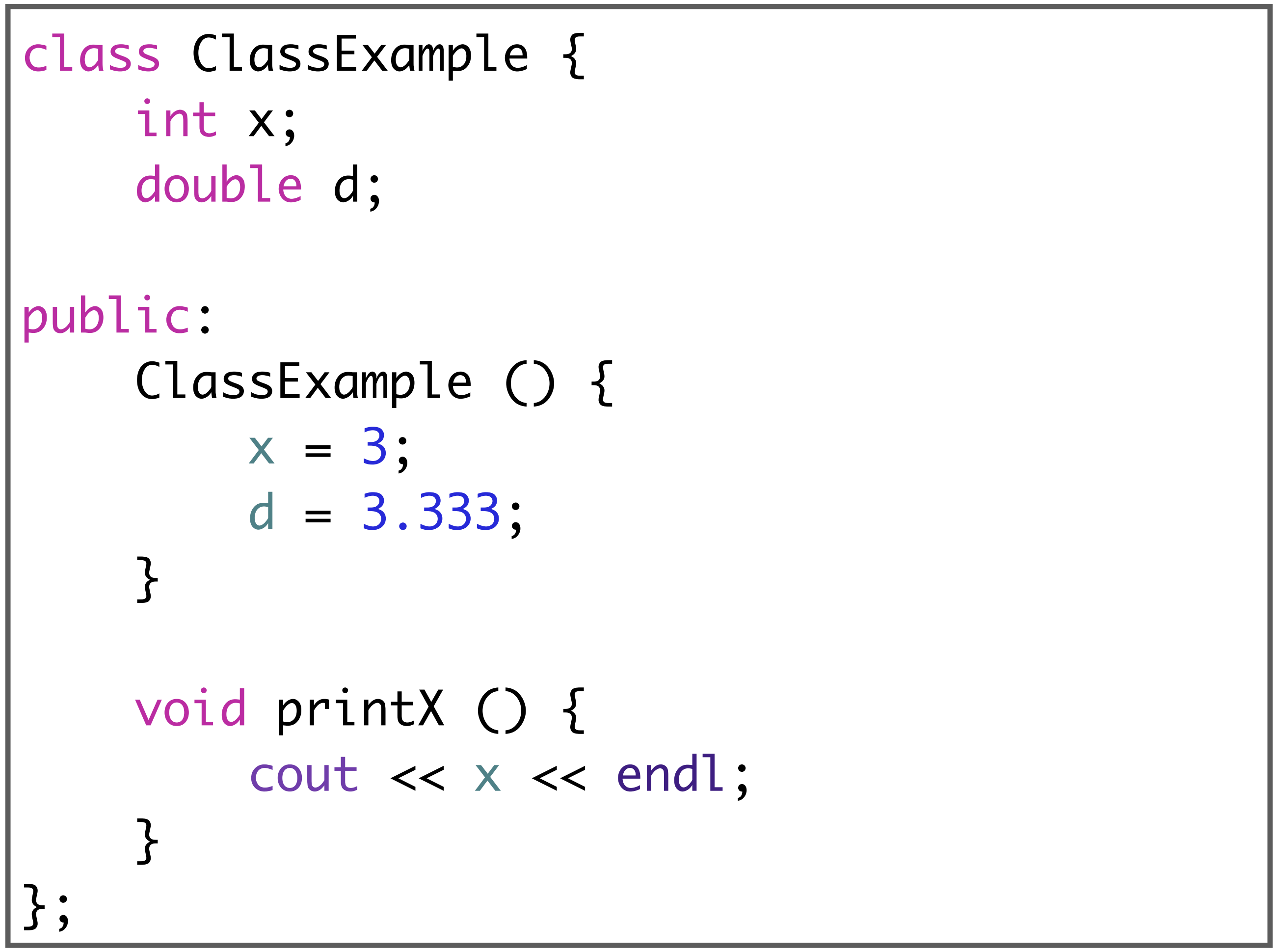
- How to give users access to the private members?
- Member functions (methods), are functions that are called by instances of a particular struct.
- Member functions called *getters* are used to allow users access to viewing private member values.
- Member functions called *setters* are used to allow users to change private member values.

# Structure: public & private tags example

```
struct Ford {
public:
    // Need function prototype
    int getFlat();
    void setFlat(int t);

    // Constructor
    Ford () {
        tires = 5;
    }

private:
    int tires;
};

// Member function definition
int Ford::getFlat () {
    return tires;
}

void Ford::setFlat (int t) {
    tires = t;
}
```

# Class: a premier

- a class and a struct are the same thing in C++ [unlike C#]
- the only difference: instead of having all members default to public access, all members of classes default to private.



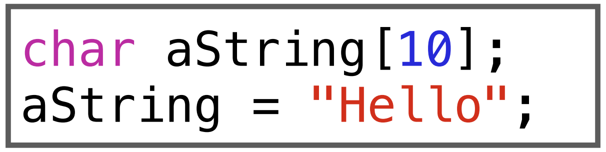
```
class ClassExample {
    int x;
    double d;

public:
    ClassExample () {
        x = 3;
        d = 3.333;
    }

    void printX () {
        cout << x << endl;
    }
};
```

# Clarifications: strcpy v.s. strncpy

- `char* strcpy (char* dst, const char* src);`
  - Copies the *C string* pointed by source into the array pointed by destination, *including the terminating null character* (and stopping at that point).
- `char* strncpy (char* dst, const char* src, size_t num);`
  - Copies the first num characters of source to destination.
  - If the end of the source C string (which is signaled by a null-character) is found before num characters have been copied, destination is padded with zeros until a total of num characters have been written to it.
  - *No null-character is implicitly appended at the end of destination if source is longer than num.* Thus, in this case, destination shall not be considered a null terminated C string (reading it as such would overflow).

# Clarifications: assignment “=” in array

- C-string values and C-string variables are not like values and variables of other data types, and many of the usual operations do not work for C-strings:
  - You *cannot* use a C-string variable in an assignment statement using =.
  - If you use == to test C-strings for equality, you will not get the result you expect. The reason for these problems is that C-strings and C-string variables are arrays.
  - Assigning a value to a C-string variable is not as simple as it is for other kinds of variables. The following is illegal:



```
char aString[10];
aString = "Hello";
```

*Illegal!*

- Although you can use the equal sign to assign a value to a C-string variable when the variable is declared, you cannot do it anywhere else in your program.
- Technically, the use of the equal sign in a declaration is an initialization, not an assignment.
- If you want to assign a value to a C-string variable, you must do something else, e.g. use predefined function strcpy as shown below: strcpy(aString, "Hello");