In C++, assignment = copying "bit-by-bit" (memberwise copying)

```
int x = 1;
string s = "ABCDE";

int y;
string t;

y = x; // copying x
t = s; // copying s
```

In the language C, assignment always meant simple memory copying bit-by-bit. In C++, it is not always that simple, however the language tries to adhere to the same behavior most of the time.

Example: Copying a structure

```
struct Data {
  int n;
};

int main() {
  Data x = {10};
  Data y;

  y = x; // get a copy of x

  cout << "x contains " << x.n << endl;     // 10
  cout << "y contains " << y.n << endl;     // 10

  y.n = 15;

  cout << "x contains " << x.n << endl;     // 10
  cout << "y contains " << y.n << endl;     // 15
}
```

Example: Copying a vector

```
vector<int> x;
x.push_back(1);
x.push_back(2);
x.push_back(3);

vector<int> y;

y = x; // get a copy of x

cout << "x contains "; print_vector(x);   // 1 2 3
cout << "y contains "; print_vector(y);   // 1 2 3
cout << endl;

y[1] = 1000;

cout << "x contains "; print_vector(x);   // 1 2 3
cout << "y contains "; print_vector(y);   // 1 1000 3
```

Returning a value from a function makes a copy

```
struct Data {
  int arr[10];
};

int main() {
  Data x;
  x = generate();
}

// Create a structure
Data generate() {
  Data d;
  for (int i = 0; i < 10; i++) {
    d.arr[i] = i*i;
  }
  return d;
}
```

Returning a value from a function makes a copy

```
struct Data {
  int arr[1000000];
};

int main() {
  Data x;
  x = generate();
}

// Create a structure
Data generate() {
  Data d;
  for (int i = 0; i < 1000000; i++) {
    d.arr[i] = i*i;
  }
  return d;     // Fine, but we don't want to waste memory!
}
```

Pointers

A pointer to a variable is the address in the memory of that variable.

```
int ten = 10;          // declare an integer variable
int *p = &ten;         // get a pointer to the variable


cout << p << end;      // print the pointer (memory address)


cout << *p << endl;    // print the value the pointer points to
                       // (dereferencing the pointer p)
```

&x is the address of the variable x

***p** dereferences the pointer p
(returns the value the pointer p points at)

Pointers

```
string s = "ABCDE";
cout << s << endl << endl;

string *p = &s;
string *p2 = p;

(*p) [1] = '.';
(*p2)[3] = '/';

cout << s << endl;
cout << *p << endl;
cout << *p2 << endl << endl;

cout << &s << endl;
cout << p << endl;
cout << p2 << endl;
```

Pointers

```
string s = "ABCDE";
cout << s << endl << endl;    // ABCDE

string *p = &s;
string *p2 = p;

(*p) [1] = '.';
(*p2)[3] = '/';

cout << s << endl;            // A.C/E
cout << *p << endl;           // A.C/E
cout << *p2 << endl << endl;  // A.C/E

cout << &s << endl;           // 0x7ffca1226ac0
cout << p << endl;            // 0x7ffca1226ac0
cout << p2 << endl;           // 0x7ffca1226ac0
```

All three pointers, `&s`, `p`, and `p2` point to the same thing.

Can we return a pointer from a function?

```
struct Data {
  int arr[1000000];
};

int main() {
  Data x;
  x = generate();
}

Data *generate() {
  Data d;
  for (int i = 0; i < 1000000; i++) {
    d.arr[i] = i*i;
  }
  return &d; // yes technically we can return this pointer
}            // but the variable d gets 'destroyed' when
             // you leave the function
```

Call stack

```
int main() {
  int arr[5];
  arr[0] = one();
  arr[1] = two();

  cout << arr[0] << endl;
  cout << arr[1] << endl;
}

int one() {
  return 1;
}

int two() {
  int y = one();
  int z = one();
  return y + z;
}
```

Allocating large data structures in the stack

This is system dependent, but for example on some computer the following program may work fine:

```
int main() {
  int arr[1000][1000];
  arr[0][0] = 1;
}
```

But the next one will crash (on Linux reporting Segmentation fault):

```
int main() {
  int arr[2000][1000];
  arr[0][0] = 1;
}
```

Allocating in the heap

- if we need to allocate a lot of memory (for example large arrays)
- if we want to create and return a big object from a function, and making additional copies is not an option. (references do mitigate the issue in C++)

```
double *pd = new double; // allocated in the heap
                         // using the operator new

*pd = 1.234;    // assign some value

cout << *pd;    // 1.234

delete pd;      // release the memory
```

Static, automatic, and dynamic variables

Static variables are allocated once at the program startup, and exist until you exit the program.

Automatic variable is a local variable declared within a block of code, it is allocated and deallocated automatically when program flow enters and leaves the block where the variable is declared.

→ *allocated in the stack*

Dynamic variables allocated by request, and not removed automatically. Have to be deleted when not needed.

→ *allocated in the heap*

Correctly returning a pointer from a function

```
struct Data { int arr[1000000]; };

int main() {
  Data *pd = generate();           // get a pointer
  cout << (*pd).arr[5] << endl;    // 25
  cout << (*pd).arr[100] << endl; // 10000

  delete pd;                       // release the memory
}

Data *generate() {
  Data *p = new Data;
  for (int i = 0; i < 1000000; i++) {
    (*p).arr[i] = i*i;
  }
  return p;
}
```