C++ For C Coders 4

Data Structures
C++ for C Coders

한동대학교 김영섭교수
idebtor@gmail.com

Arrays
Structures
Classes

Array

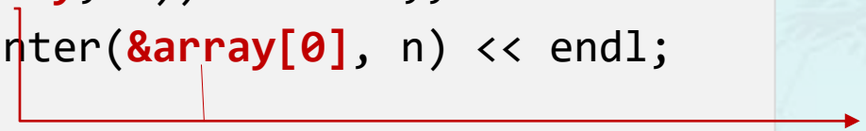
- Array is a collection of data of the same type.
- Why array?
  - **Efficient random access** (constant time) but inefficient insertion and deletion of elements.
  - **Good locality** of reference when iterating through - much faster than iterating through (say) a linked list of the same size, which tends to jump around in memory.
  - Consequently, arrays are most appropriate for storing a fixed amount of data which will be accessed in an unpredictable fashion

Array

- Array is a collection of data of the same type.
- Array in C/C++
  - **base address:**
    It is the address of the first element of an array which is `&list[0]` or `list`.
  - **pointer arithmetic:**
    `(ptr + 1)` references to the next element of array regardless of its type.
  - **dereferencing operator ***
    `*(ptr + i)` indicates contents of the `(ptr + i)` position of array.

Array

```
void main(void) {
  double array[] = {0, 1, 2, 3, 4};
  int n = sizeof(array) / sizeof(array[0]);
  cout << sum(array, n)) << endl;;
  cout << sumPointer(&array[0], n) << endl;
}
```



Array

```
void main(void) {
  double array[] = {0, 1, 2, 3, 4};
  int n = sizeof(array) / sizeof(array[0]);
  cout << sum(array, n)) << endl;;
  cout << sumPointer(&array[0], n) << endl;
}
```

equivalent

```
double sum(double a[], int n) {
  double total = 0;

  for (int i = 0; i < n; i++)
    total += a[i];
  return total;
}
```

```
double sumPointer(double a[], int n) {
  double total = 0;

  for (int i = 0; i < n; i++, a++)
    total += *a;
  return total;
}
```

Array

```
void main(void) {
  double array[] = {0, 1, 2, 3, 4};
  int n = sizeof(array) / sizeof(array[0]);
  cout << sum(array, n)) << endl;;
  cout << sumPointer(&array[0], n) << endl;
}
```

equivalent

```
double sum(double a[], int n) {
  double total = 0;

  for (int i = 0; i < n; i++)
    total += a[i];
  return total;
}
```

```
double sumPointer(double a[], int n) {
  double total = 0;

  for (int i = 0; i < n; i++, a++)
    total += *a;
  return total;
}
```

```
double sumPointer(double a[], int n) {
  double total = 0;

  for (int i = 0; i < n; i++)
    total +=
  return total;
}
```

Struct

- **Struct** is a handy way to organize data of the **different type**.
- Like **class** (actually the idea of class in OOP is derived from **struct**), provide encapsulation of data, it handles a group of data as a whole.
- The **struct** keyword defines a structure type followed by an identifier (name of the structure). Then inside the curly braces, you can declare one or more members of that structure.

C

```
struct Car{
   int  age;
   char tag[32];
};

struct Car one;
one.age = 21;
strcpy(one.tag, "sky");
```

member access operator

C

C++

Struct

- The **typedef** is used to give a data type a new name.
  It is mostly done in order to make the code cleaner.
- Keyword **typedef** can be used to simplify syntax of a structure in C.

C

```
struct Car{
   int  age;
   char tag[32];
};

struct Car one;
one.age = 21;
strcpy(one.tag, "sky");
```

C

```
typedef struct Car{
   int  age;
   char tag[32];
} Car;

Car one;
one.age = 21;
strcpy(one.tag, "sky");
```

C++

member access operator

Struct

- The **typedef** is used to give a data type a new name.
  It is mostly done in order to make the code cleaner.
- Keyword **typedef** can be used to simplify syntax of a structure in C.

- In C++, you can do the same thing **without typedef** and more.

C

```
struct Car{
   int  age;
   char tag[32];
};

struct Car one;
one.age = 21;
strcpy(one.tag, "sky");
```

C

```
typedef struct Car{
   int  age;
   char tag[32];
} Car;

Car one;
one.age = 21;
strcpy(one.tag, "sky");
```

C++

```
struct Car{
   int  age;
   string tag;
};

Car one;
one.age = 21;
one.tag = "sky";
```

member access operator

Using pointer with struct

```
struct Car{
    int    age;
    string tag;
};
Car  ur = {25, "cat"};
Car *my = (Car *)malloc(sizeof(Car));
          = "sky";
          = 20;
```

C++

Recall a pointer can store
only a address of memory.

Using pointer with struct

```
struct Car{
    int    age;
    string tag;
};
Car  ur = {25, "cat"};
Car *my = (Car *)malloc(sizeof(Car));
(*my).tag = "sky";
(*my).age = 20;
```

C++

Recall a pointer can store
only a address of memory.

Using pointer with struct

```
struct Car{
    int    age;
    string tag;
};
Car  ur = {25, "cat"};
Car *my = (Car *)malloc(sizeof(Car));
(*my).tag = "sky";
(*my).age = 20;
```

C++

```
struct Car{
    int    age;
    string tag;
};
Car  ur = {25, "cat"};
Car *my = (Car *)malloc(sizeof(Car));
my->tag = "sky";
my->age = 20;
```

member access operator

Using pointer with struct

```
struct Car{
    int    age;
    string tag;
};
Car  ur = {25, "cat"};
Car *my = (Car *)malloc(sizeof(Car));
(*my).tag = "sky";
(*my).age = 20;
```

C++

```
struct Car{
    int    age;
    string tag;
};
Car  ur = {25, "cat"};
Car *my = (Car *)malloc(sizeof(Car));
my->tag = "sky";
my->age = 20;
```

member access operator

```
struct Car{
    int    age;
    string tag;
};
Car  ur = {25, "cat"};
Car *my = new Car {20, "sky"};
```

struct initialization

Passing a pointer to a function

C++

```
struct Car{
    int    age;
    string tag;
};

bool older(Car *a, Car *b) {
    return a->age > b->age ;
};

int main() {
  Car  ur = {25, "cat"};
  Car *my = new Car {20, "sky"};
  cout << "ur age: " << ur.age << endl;
  cout << "my age: " << my->age << endl;
  cout << "ur older? " << older(________) << endl;
  return 0;
}
```

Passing a pointer to a function

```
struct Car{
    int    age;
    string tag;
};

bool older(Car *a, Car *b) {
    return a->age > b->age ;
};

int main() {
  Car  ur = {25, "cat"};
  Car *my = new Car {20, "sky"};
  cout << "ur age: " << ur.age << endl;
  cout << "my age: " << my->age << endl;
  cout << "ur older? " << older(&ur, my) << endl;
  return 0;
}
```

C++

Do you see a bug in the code above?

Passing a pointer to a function

C++

```
struct Car{
    int    age;
    string tag;
};

bool older(Car *a, Car *b) {
    return a->age > b->age ;
};

int main() {
  Car  ur = {25, "cat"};
  Car *my = new Car {20, "sky"};
  cout << "ur age: " << ur.age << endl;
  cout << "my age: " << my->age << endl;
  cout << "ur older? " << older(&ur, my) << endl;
  delete my;
  return 0;
}
```

Using using

C++

```
struct Car{
    int    age;
    string tag;
};

int main() {
  Car  ur = {25, "cat"};
  Car *my = new Car {20, "sky"};

  // copy my contents to ur
  ur = *my;
  delete my;
  return 0;
}
```

Let's go one more step!

- Redefine `Car *` using `using`.

Using using

C++

```
struct Car{
    int    age;
    string tag;
};

int main() {
  Car  ur = {25, "cat"};
  Car *my = new Car {20, "sky"};

  // copy my contents to ur
  ur = *my;
  delete my;
  return 0;
}
```

```
struct Car{
    int    age;
    string tag;
};
using pCar = Car *;
int main() {
  Car  ur = {25, "cat"};
  pCar my = new Car {20, "sky"};

  // copy my contents to ur
  ur = *my;
  delete my;
  return 0;
}
```

Let's go one more step!

- Redefine `Car *` using `using`.

Quiz: Rewrite the code using C++ reference

C++

```
struct Car{
    int    age;
    string tag;
};

bool older(Car *a, Car *b) {
    return a->age > b->age ;
};

int main() {
  Car  ur = {25, "cat"};
  Car *my = new Car {20, "sky"};
  cout << "ur age: " << ur.age << endl;
  cout << "my age: " << my->age << endl;
  cout << "ur older? " << older(&ur, my) << endl;
  delete my;
  return 0;
}
```

```
struct Car{
    int    age;
    string tag;
};

bool older(            ) {
    return ______________________;
};

int main() {
  Car  ur = {25, "cat"};
  Car  my = {20, "sky"};
  cout << "ur age: " << __________ << endl;
  cout << "my age: " << __________ << endl;
  cout << "ur older? " << older(________) << endl;

  return 0;
}
```

Struct vs. Class in C++

- The **member variables** and methods are hidden from the outside world, **unless** their declaration follows a **public** label. [*encapsulation*]
- There can be a pair of special methods – the **constructor** and **destructor** – that are run automatically when **an instance of the class** [an object] is created and destroyed.
- Operators to work on the new data type can be defined using special methods [**member functions; methods**].
- One **class** can be used as the basis for the definition of another [inheritance].
- Declaring a variable of the new type [an instance of the class; an object] requires just the name of the **class** – the keyword **class** is not required.

Struct vs. Class in C++

- The **member variables** and methods are hidden from the outside world, **unless** their declaration follows a **public** label. [*encapsulation*]
- There can be a pair of special methods – the **constructor** and **destructor** – that are run automatically when **an instance of the class** [an object] is created and destroyed.
- Operators to work on the new data type can be defined using special methods [**member functions; methods**].
- One **class** can be used as the basis for the definition of another [inheritance].
- Declaring a variable of the new type [an instance of the class; an object] requires just the name of the **class** – the keyword **class** is not required.

◈ Believe it or not, the *only* difference between a **struct** and **class** in C++ is the default accessibility of member variables and methods.
In a **struct** they are public by default;
In a **class** they are private.

C++ For C Coders 4

Data Structures
C++ for C Coders

한동대학교 김영섭교수
idebtor@gmail.com

Arrays
Structures
Classes