IM 1003: Programming Design

Self-defined data types (in C)

Ling-Chieh Kung

Department of Information Management
National Taiwan University

March 31, 2014

Self-defined data types

- We can define data types by ourselves.
  - By **combining** data types into a composite type.
  - By **redefining** data types.
- We can always complete every program without self-defined data types.
  - But we can make our program **clearer** and more **flexible** by using them.
- In C, there are many ways of creating self-defined data types.
  - `typedef`, `struct`, `union`, and `enum`.
  - We will introduce only the first two.
  - You can learn the other two by yourself (or ignore them at this moment).

Outline

- struct
- typedef
- struct with member functions

Example

- How to write a program to create two points *A* and *B* on the Cartesian coordinate system, compute vector *AB*, and print it out?
  - Let's implement a function that computes the vector.

```
int main()
{
  int x1 = 0, x2 = 0;
  int y1 = 10, y2 = 20;
  int rx = 0, ry = 0;
  vector (x1, y1, x2, y2, rx, ry);
  cout << rx << " " << ry << endl;
  return 0;
}
```

```
void vector(int x1, int y1, int x2,
  int y2, int& rx, int& ry)
{
  rx = x2 - x1;
  ry = y2 - y1;
}
```

  - May we improve the program?

struct

- There are so many variables!
  - Some of them must be used in pairs.
- We want to **group** different data types into a single type.
  - Group **x** and **y** into a "point".
- In C, we do so by using **struct** (abbreviation of structure).
  - We may group basic data types, nonbasic data types (e.g., pointers and arrays), or even self-defined data types.
  - We do so when an item naturally consists of multiple **attributes**.
  - We do so to make the program easier to read and maintain.

Example with struct

- Let's define a **new type** **Point**:

```
struct Point
{
  int x;
  int y;
};
```

  - The keyword **struct** is used to define structures.
- Now it is a data type and we can use it to **declare variables**.

Example with struct

- With the new data type, the program can now be written in this way:
  - **Declare** variables with the self-defined type name.
  - **Assign** values to both attributes by grouping values by curly brackets.
  - **Access** attributes through the **dot operator**.
- The function is also changed:
  - Use **Point** as a parameter type.
  - No need to call by reference.

```
Point vector (Point a, Point b)
  // Point as parameters
{
  Point vecXY;
  vecXY.x = B.x - A.x;
  vecXY.y = B.y - A.y;
  return vecXY; // return a Point
}
int main()
{
  Point a = {0, 0}, b = {10, 20};
  Point vecAB = vector(a, b);
  cout << vecAB.x << " ";
  cout << vecAB.y << endl;
  return 0;
}
```

struct definition

- The syntax of defining a structure is:
  - A structure is typically named with the first letter capitalized.
  - An attribute/field can be of a basic data type, a nonbasic data type, or a self-defined data type.
  - The number of attributes is unlimited.
  - All those semicolons are required.
- As another example, let's add one more attribute into **Point**:

```
struct struct name
{
  type1 field 1;
  type2 field 2;
  type3 field 3;
  // more fields
};
```

```
struct Point
{
  int x;
  int y;
  char name;
};
```

struct variable declaration

- To declare a variable defined as a structure, use

  struct name variable name;

  - `Point A;`
  - `Point B, C, thisIsAPoint;`
  - `Point staticPointArray[10];`
  - `Point* pointPtr = &aPoint;`
  - `Point* dynamicPointArray = new Point[10];`
- You may also (but usually people do not) write
  - `struct Point A;`
  - `struct Point B, C, thisIsAPoint;`

Accessing struct attributes

- Use the dot operator "." to access a **struct** variable's attributes.

struct variable*.*attribute name

  - An attribute is a single variable.
  - We may do all the regular operations on an attribute.

```
Point a, b;
a.x = 0; // assignment
a.y = a.x + 10; // arithmetic
a.name = 'A';
cin >> b.name; // input
cout << a.x; // print out
b.y = a.y; // assignment
```

struct assignment

- We may use curly brackets to assign values to multiple attributes.

```
Point A = {0, 0, 'A'};
Point B;
B = {10, 20, 'B'};
C = {5, 0};
D = {2};
```

- **Partial assignments** are allowed (with unassigned attributes set to 0).

```
struct Point
{
  int x;
  int y;
  int z;
};

int main()
{
  Point A[100];
  for (int i = 0; i < 50; i++)
    A[i] = {20};
  for (int i = 0; i < 100; i++)
    cout << A[i].y << " " << A[i].z << endl;
  return 0;
}
```

struct and functions

- You may pass a **struct** variable as an argument into a function.
- You may return a **struct** variable from a function, too.
- Passing a **struct** variable by default is a call-by-value process.
- You may call by reference, as always.

```
struct Point
{
  int x;
  int y;
};
void reflect (Point& a)
{
  int temp = a.x;
  a.x = a.y;
  a.y = temp;
}
```

```
int main()
{
  Point a = {10, 20};
  cout << a.x << " "
       << a.y << endl;
  reflect (a);
  cout << a.x << " "
       << a.y << endl;
  return 0;
}
```

Memory allocation for `struct`

- When we declare a structure variable, how does the compiler allocate memory spaces to it?
  - How many bytes are allocated in total?
  - Are attributes put together or separated?
  - What if we declare a structure array?

```
struct Point
{
  int x;
  int y;
};
int main()
{
  Point a[10];
  cout << sizeof (Point) << " " << sizeof (a) << endl;
  cout << &a << endl;
  for (int i = 0; i < 10; i++)
    cout << &a[i] << " " << &a[i].x << " " << &a[i].y << endl;
  Point* b = new Point[20];
  cout << sizeof (b) << endl;
  delete [] b;
  b = NULL;
  return 0;
}
```

Outline

- struct
- typedef
- struct with member functions

typedef

- **typedef** is the abbreviation of "**type definition**".
- It allows us to create a new data type **from** another data type.
- To write a type definition statement:

```
typedef old_type new_type;
```

- This defines ***new_type*** as ***old_type***.
  - ***old_type*** must be an existing data type.
- So we do not really create any new type. Why do we do so?

Example

- Suppose we want to write a program that converts a given US dollar amount into an NT dollar amount.

```
double nt = 0;
double us = 0;
cin >> us;
nt = us * 29;
cout << nt << endl;
```

- Suppose in your program there are ten different kinds of monetary units, and you declared all of them to be **double**.
- What if one day you want to change all the types to **float**?

Example with typedef

- To avoid modifying ten declaration statements, **typedef** helps!

```
typedef double Dollar; // define Dollar as double
Dollar nt; // declare a variable as Dollar
Dollar us;
cin >> us;
nt = us * 29;
cout << nt << endl;
```

  - **Dollar** is a self-defined data type. It can be used to declare variables.
  - If one day we want to change the type into **float**, **int**, etc., we only need to do one modification.
  - Also, when one looks at your program, she will know that **nt** and **us** are "dollars" instead of just some double variables.

"Type" life cycle

- You can put the **`typedef`** statement anywhere in the program.
  - At the beginning of the program, in the main function, inside a block, etc.
- The self-defined type can be used only **in the block** (if you declare it in any block).
- The same rule applies to **`struct`**.

Example

- What may happen if we compile this program?
- How to fix it?
- Put the type definition statements and structure definition in the place that anyone can find it easily.
  - Usually it is the beginning of the program, just under the include statement.
- Put them globally unless you really use them locally.

```
int exchange (Dollar from, double rate);
int main()
{
  typedef double Dollar;
  Dollar NT, US;
  cin >> US;
  NT = exchange (US, 29);
  cout << NT << endl;
  return 0;
}
int exchange (Dollar from, double rate)
{
  return from * rate;
}
```

typedef from struct

- Recall that we have done the following:

```
Point a = {0, 0};
Point b = {10, 20};
Point vecAB = vector (a, b);
```

  - But **vecAB** is not a point! It is a vector.
  - Vectors have the same attributes as points do. Should we define another structure that is identical to **Point**?
- We may combine **typedef** and **struct**.

```
// define Vector from Point
typedef Point Vector;
```

typedef from struct

- Suppose we do this:

```
struct Point
{
  int x;
  int y;
}; // end of struct definition
// define from struct
typedef Point Vector;
```

- Then we may write:

```
Point a(0, 0);
Point b(10, 20);
Vector vecAB = vector (a, b);
```

Example: <ctime>

- Many **C++ standard library** functionalities are provided with new types defined by `typedef`.
- As an example, the function `clock()`, defined in `<ctime>`, returns the number of system clock ticks elapsed since the execution of the program.
- What is `clock_t`?

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

int main()
{
  clock_t sTime = clock();
  for(int i = 0; i < 1000000000; i++)
    ;
  clock_t eTime = clock();

  cout << sTime << " " << eTime << endl;
  return 0;
}
```

Example: <ctime>

- **clock()** returns a type **clock_t** variable (for the number of ticks).
  - **clock_t** is actually a **long int**. In **<ctime>**, there is a statement:

```
typedef long int clock_t;
```

- So in our own functions, we may write **clock_t sTime = clock();**.
  - We may change it to **long int sTime = clock();**.
- Why does the standard library do so?
- To print out the number of seconds instead of ticks:

```
cout << static_cast<double>(eTime - sTime) / CLOCKS_PER_SEC << endl;
```

Outline

- `struct`
- `typedef`
- `struct` with member functions

Member variables

- Recall that we have defined

```
struct Point
{
  int x;
  int y;
};
```

  - We say that **x** and **y** are the attributes or fields of the structure **Point**.
  - They are also called the **member variables** of **Point**.
- Suppose we want to write a function that calculate a given point's distance from the origin. How may we do this?

A global-function implementation

- We may write a function which takes a point as a parameter:

```
double distOri(Point p)
{
  double dist = sqrt(pow(p.x, 2) + pow(p.y, 2));
  return dist;
}
```

  - We need to include **`<cmath>`**.
- This works, but this function is doing something that is related to **only one** point.
  - And it is calculating a **property** of the point.
- We may want to write this function as a part of **`Point`**.

A member-function implementation

- We may redefine **Point** to include a **member function**:
  - **distOri()** is a member function of **Point**.
  - **No argument** is needed.
  - **Who's x** and **y**?
- To invoke a member function:

```
int main()
{
  Point a = {3, 4};
  cout << a.distOri();
  return 0;
}
```

```
struct Point
{
  int x;
  int y;
  double distOri()
  {
    double dist = sqrt(pow(x, 2) + pow(y, 2));
    return dist;
  }
};
```

A member-function implementation

- One may define a member function outside the **struct** statement.

```
struct Point
{
  int x;
  int y;
  double distOri ();
};
double Point::distOri () // scope resolution
{                        // is required
  double dist = sqrt (pow (x, 2) + pow (y, 2));
  return dist;
}
```

  – In fact this is typically preferred. Why?

The two ways of thinking

- What is the difference between the global-function and member-function implementations?
- The perspectives of looking at this functionality is different.
  - As a global function: I want to **create a machine** outside a point. Once I throw a point into it, I get the desired distance.
  - As a member function: I want to **attach an operation** on a point. Once I run this operation, I get the desired distance.
- The second perspective is preferred when we design more complicated items.
- The second way also enhances **modularity**.

Another example

- Recall that we have written a **`reflect`** function:

```
struct Point
{
  int x;
  int y;
};
void reflect (Point& a)
{
  int temp = a.x;
  a.x = a.y;
  a.y = temp;
}
```

```
int main()
{
  Point a = {10, 20};
  cout << a.x << " "
       << a.y << endl;
  reflect (a);
  cout << a.x << " "
       << a.y << endl;
  return 0;
}
```

- May we (should we) implement it as a member function?

Another example

- A member-function implementation:

```
struct Point
{
  int x;
  int y;
  void reflect();
};
void Point::reflect()
{
  int temp = x;
  x = y;
  y = temp;
}
```

```
int main()
{
  Point a = {10, 20};
  cout << a.x << " "
       << a.y << endl;
  a.reflect();
  cout << a.x << " "
       << a.y << endl;
  return 0;
}
```

- Which one do you prefer?

One common "error" for beginners

- What is "wrong" in the following definition?

```
struct Point
{
  int x;
  int y;
  double distOri (Point p);
};
double Point::distOri (Point p)
{
  double dist = sqrt (pow (p.x, 2) + pow (p.y, 2));
  return dist;
}
```

- The program can still run. However, never do this!

One common "error" for beginners

- How about this?

```
struct Point
{
  int x;
  int y;
  void reflect (Point& p);
};
void Point::reflect(Point& p)
{
  int temp = p.x;
  p.x = p.y;
  p.y = temp;
}
```