IM 1003: Computer Programming
Self-defined Data Types in C

Ling-Chieh Kung

Department of Information Management
National Taiwan University

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`, `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

`struct`

- You can **group** some different data types into a single type by using `struct`.
  - `struct` is the abbreviation of structure.
  - We may group basic data types.
  - We may also group nonbasic data types (e.g., pointers and arrays).
  - We may even group self-defined data types in another self-defined data type.
- Some items naturally consists of multiple **attributes**.
  - These attributes mean something only if they appear together.
  - It will be better to group them into a single data type.

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?
  - Implement a function that computes the vector.

```
void vector (int x1, int y1, int x2,
  int y2, int& rx, int& ry);
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 avoid using call by reference?

Example with `struct`

- What are the drawbacks of this program?
  - There are so many variables.
  - Variables must be used in pairs (e.g., **`x1`** and **`y1`**).
- It will be easier to develop and maintain our program if we can create a new type which contains both the *x*- and *y*-coordinate.
- In C, this can be done by defining a **structure**.
  - The keyword **`struct`** is used to define structures.
- Now it is a data type and we can use it to **declare variables**.
  - At those places after its definition.

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


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**.

```
#include <iostream>
using namespace std;
struct Point
{
  int x;
  int y;
};
int main()
{
  Point A = {0, 0}, B = {10, 20};
  Point vecAB = vector(A, B);
  cout << vecAB.x << " ";
  cout << vecAB.y << endl;
  return 0;
}
```


Example with `struct`

- With the new data type, the function can now be implemented in this way:

```
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
}
```

  - The function is easier to read and understand.
  - No need of call by reference.

`struct` definition

- The syntax of defining a structure is:
  - A structure is typically name 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;
  // ...
};
```

```
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;`** // name variables in the usual way
  - **`Point staticPointArray[10];`**
  - **`Point* pointPtr = NULL;`**
  - **`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
```

- We may view an attribute as 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 assignment is allowed.
  - Uninitialized attributes may be anything, even if part of the attributes are given values.
  - Example "**09_01_structInit**".

`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 implement call by reference in the usual way.
  - Example "**09_02_structFunc**".
- You may pass or return one of the attributes, just as a single variable.

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?
- Example "**09_03_structMemory**".
- The memory allocation mechanism will be discussed again when we talk about "class inheritance".

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.
  - Typically from a basic 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. What is the point of doing 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. For example, at the beginning of the program, the main function, or inside any block.
- 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?

```
int exchange (Dollar from, Double rate);
int main()
{
  typedef double Dollar;
  Dollar NT, US;
  cin >> US;
  NT = exchange(US, 40);
  cout << NT << endl;
  return 0;
}
int exchange (Dollar from, Double rate)
{
  return from * rate;
}
```

Good programming style

- 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.

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.
  - But vectors have the same attributes as points do. Should we define another structure that is identical to **Point**?
- We may combine **typedef** and **struct**.

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);
```


C++ standard library

- You may not use **typedef** in the future. However, at least you have to know what it is.
- Many C++ standard library functionalities are provided with new types defined by **typedef**.
  - As an example, the function **clock()**, defined in **<ctime>**, returns a type **clock_t** variable.
  - **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 stTime = clock();**.
  - Why does the standard library do so?

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;
}
```

  - Certainly we need to include **`<cmath>`**.
- This works, but this function is doing something that is related to **only one** 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**:

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

  - **`distOri()`** is a member function of **`Point`**.
  - Note that **no argument** is needed.
  - Note how it accesses the two member variables. **Who's** **`x`** and **`y`**?

A member-function implementation

- To access a structure's member function, use the dot operator.

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


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?
  – They both work if they are implemented correctly.
- 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. You will start to experience this after the introduction of classes.
- The second way also enhances **modularity**.

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;
}
```

- It does not generate a compilation error. However, never do this!

A very brief introduction of classes

- While in C we use structure, in C++ we use structures and **classes**.
  - A class is a self-defined data type.
  - Conceptually similar to a structure: A class can have its own **member variables** and **member functions**.
  - Variables declared with classes are called **objects**.
- However, classes are more powerful than structures.
- One main difference is that members defined in a class can be categorized into different "**privacy levels**":
  - One may **control the access** of its members.
  - `private`, `protected`, and `public`.

Access control of class members

- Members defined in a class are by default **private**.
  - A private member can only be accessed by its own member function.

```
class Point
{
  int x;
  int y;
  double distOri ();
};
double Point::distOri ()
{
  double dist = sqrt (pow (x, 2) + pow (y, 2));
  return dist;
} // so far so good ...
```


Access control of class members

- But in the main function:

```
int main()
{
  Point a;
  cout << a.distOri(); // error!
  return 0;
}
```

- One **cannot** invoke a private member in the main function.
  - Because the main function is not a member function of **`Point`**.
- The way of initialization is also changed. This will be discussed in the future.

Access control of class members

- We need to declare members as public for them to be accessed in the main function.

```
class Point
{
  private:
    int x;
    int y;
  public:
    double distOri ();
    // no change in the definition
};
```

  - Example "**09_04_class**".
  - **`a.distOri()`** is allowed in **`main()`** but **`a.x`** and **`a.y`** are still not allowed.

Access control of class members

- Why do we want to do access control?
- Because we want to ensure that people use our classes in a **safe and controlled** way.
- This, as well as many other related ideas to enhance and efficiency of design, maintain, and extend large-scale programs, will be discussed when we introduce object-oriented programming.