

Programming Design

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**, and **enum**.
 - We will introduce only the first two.
 - You can learn the other two by yourself (or ignore them in this course).

Outline

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

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 << "\n";
    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 << "\n";
    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 an example, let's declare a structure **Point**:

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

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


struct variable declaration

- To declare a variable defined as a structure, use

```
struct name variable name;
```

- For example:

```
Point A;  
Point B, C, thisIsAPoint;  
Point staticPointArray[10];  
Point* pointPtr = &thisIsAPoint;  
Point* dynamicPointArray = new Point[10];
```

- You may also (but usually people do not) write

```
struct Point A;
```

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  
cin >> b.x; // input  
cout << a.x; // print out  
b.y = a.y; // assignment
```

struct assignment

- We may also use curly brackets to assign values to multiple attributes.
 - **Partial assignments** are allowed (with unassigned attributes set to 0).

```
Point A = {0, 0, -8};  
Point B;  
B = {10, 20, 5};  
C = {5, 0};  
D = {2};
```

```
struct Point {  
    int x;  
    int y;  
    int z;  
};  
int main() {  
    Point A[100];  
    for (int i = 0; i < 50; i++)  
        A[i] = {i};  
    for (int i = 0; i < 100; i++)  
        cout << A[i].x << " " << A[i].y  
            << " " << A[i].z << "\n";  
    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 << "\n";
    reflect (a);
    cout << a.x << " "
         << a.y << "\n";
    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?

Memory allocation for struct

```
struct Point {
    int x;
    int y;
};
int main() {
    Point a[3];
    cout << sizeof(Point) << " "
         << sizeof(a) << "\n";
    cout << &a << "\n";
    for (int i = 0; i < 3; i++)
        cout << &a[i] << " " << &a[i].x
             << " " << &a[i].y << "\n";
    Point* b = new Point[3];
    cout << sizeof(b) << "\n";
    delete [] b;
    b = NULL;
    return 0;
}
```

Outline

- **struct**
- **typedef**
- **struct** with member functions
- Randomization

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 that 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 << "\n";
```

- 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 << "\n";
```

- **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 << "\n";
    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;
```

```
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 << "\n";
    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();` (but please do not).
 - Why does the standard library do so? Why should we use `clock_t`?
- To print out the number of seconds instead of ticks:

```
cout << static_cast<double>(eTime - sTime) / CLOCKS_PER_SEC << "\n";
```

Example: <cstring>

- Recall that many functions are defined in <cstring>.

- The function **strlen** actually returns **size_t**.

```
size_t strlen(const char* str);
```

- Therefore, it is suggested to use **size_t** also in the caller.

```
void reverse(char p[])
{
    size_t n = strlen(p);
    char* temp = new char[n];
    for(int i = 0; i < n; i++)
        temp[i] = p[n - i - 1];
    for(int i = 0; i < n; i++)
        p[i] = temp[i];
    delete [] temp;
}
```

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

void reverse(char* p);
int main()
{
    char s[100] = "12345";
    reverse(s);
    return 0;
}
```


Outline

- `struct`
- `typedef`
- **`struct with member functions`**
- Randomization

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()
    {
        return sqrt(pow(x, 2) + pow(y, 2));
    }
};
```

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
    return sqrt(pow(x, 2) + pow(y, 2));
}
```

- In fact this is typically preferred. Why?

Two different perspectives

- 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 main()
{
    int x;                  {
    int y;                  Point a = {10, 20};
};                          cout << a.x << " "
void reflect(Point& a)      << a.y << "\n";
{                            reflect(a);
    int temp = a.x;         cout << a.x << " "
    a.x = a.y;              << a.y << "\n";
    a.y = temp;            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 << "\n";
    a.reflect();
    cout << a.x << " "
         << a.y << "\n";
    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 distOri1(Point p);
    double distOri2(Point p);
};
```

```
double Point::distOri1(Point p)
{
    double dist = sqrt(pow(x, 2) + pow(y, 2));
    return dist;
}
double Point::distOri2(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 reflect1(Point& p);
    void reflect2(Point& p);
};
```

```
void Point::reflect1(Point& p)
{
    int temp = p.x;
    p.x = p.y;
    p.y = temp;
}
void Point::reflect2(Point& p)
{
    int temp = x;
    x = y;
    y = temp;
}
```

Outline

- `struct`
- `typedef`
- `struct` with member functions
- **Randomization**

Random numbers

- In some situations, we need to generate **random numbers**.
 - For example, a teacher may want to write a program to randomly draw one student to answer a question.
- In C++, randomization can be done with two functions, **srand()** and **rand()**.
- They are defined in **<cstdlib>**.

rand()

- `int rand();`
- It “randomly” returns an **integer** between 0 and **RAND_MAX** (in `<cstdlib>`, typically 32767).
- Try to run it for multiple times.
 - What happened?

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

int main()
{
    int rn = 0;
    for (int i = 0; i < 10; i++)
    {
        rn = rand();
        cout << rn << " ";
    }
    return 0;
}
```

rand()

- **rand()** returns a “**pseudo-random**” integer.
 - They just look **like** random numbers. But they are not really random.
 - There is a formula to produce each number.
 - e.g., $r_i = (943285761 * r_{i-1} + 18763571) \bmod 32767$.
 - Given the same r_0 , all pseudo-random numbers will be the same.
- You need to have a “random number **seed**”.
 - r_0 for this example.

srand()

- We use srand to determine the seed.

```
void srand(unsigned int);
```

- A seed can be generated based on the input number.
- The sequence will become different.
- Try to run it for multiple times.
 - What happened?

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

int main()
{
    srand(0);
    int rn = 0;
    for (int i = 0; i < 10; i++)
    {
        rn = rand();
        cout << rn << " ";
    }
    return 0;
}
```

srand()

- We must give **srand()** **different arguments**.
- In many cases, we use **time(0)** to be the argument of **srand()**.
 - The function **time(0)**, defined in **<ctime>**, returns the number of seconds that have past since 0:0:0, Jan, 1st, 1970.

```
time_t time(time_t* timer);
```

- It also sets ***timer** to that number of seconds (if it is not a null pointer).
- Let's try it:

```
time_t t = srand(time(0));  
cout << t << "\n";
```

```
time_t t;  
time(&t);  
srand(t);  
cout << t << "\n";
```


srand() and time()

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

int main()
{
    srand(time(0)); // good
    int rn = 0;
    for(int i = 0; i < 10; i++)
    {
        rn = rand();
        cout << rn << " ";
    }
    return 0;
}
```

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

int main()
{
    int rn = 0;
    for(int i = 0; i < 10; i++)
    {
        srand(time(0)); // bad
        rn = rand();
        cout << rn << " ";
    }
    return 0;
}
```

Random numbers in a range

- If you want to produce random numbers in a specific range, use %.
- What is the range in this program?
- How about this?

```
rn = (static_cast<double>(rand() % 501)) / 100;
```

- More powerful random number generators are provided in `<random>` (if your compiler is new enough).

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

int main()
{
    srand(time(0));
    int rn = 0;
    for(int i = 0; i < 10; i++)
    {
        rn = ((rand() % 10)) + 100;
        cout << rn << " ";
    }
    return 0;
}
```

A self-defined random number generator

- Let's implement our own random number generator!
- A randomizer generates random numbers according to

$$r_i = (ar_{i-1} + b) \bmod c.$$

Therefore, a randomizer is characterized with attributes a , b , and c .

- It also needs r_0 , the seed, as an attribute.
- It should provide an operation that generates the next random number.

```
struct Randomizer
{
    int a;
    int b;
    int c;
    int cur;
    int rand();
};
int Randomizer::rand()
{
    cur = (a * cur + b) % c;
    return cur;
}
```

A self-defined random number generator

- To use it, first we generate a randomizer.
 - Assign appropriate attributes.
 - Invoke **rand()** repeatedly.
- Different attributes create different randomizers.
 - **r1** and **r2**, which one is better?

```
int main()
{
    Randomizer r1 = {10, 4, 31, 0};
    for(int i = 0; i < 10; i++)
        cout << r1.rand() << " ";
    cout << "\n";
    Randomizer r2 = {10, 7, 32, 0};
    for(int i = 0; i < 10; i++)
        cout << r2.rand() << " ";
    return 0;
}
```

Should I use a structure?

- Without structures, the randomization function may be

```
int badRand(int a, int b, int c, int cur)
{
    return (a * cur + b) % c;
}
```

- One needs to keep a record on **cur**.
- One needs to ensure that **a**, **b**, and **c** are the same all the time.
- The function is harder to use. The program is harder to maintain.
 - Structures enhance **modularity**!
- Next week we introduce **classes**, which are “more powerful” structures.