Ch12-Structures

August 16, 2020

1 Structures

1.1 Topics

- compound and heterogeneous types
- structrues and records types
- examples of structures and objects
- aggregate operations on objects
- passing structure to and and returing from functions
- array of structures
- structures in another structures

1.2 Compound and Heterogeneous types

- most of the data types we've worked with represent a single value
  - an interger, floating-point value, char, etc.
- we've also worked with array of similar values such as string, array of integers or array of strings
- array and string can be considered as compound types but all elements are **homogeneous (same) type**
- C++ possibly can't provide all the types of data that programs need to efficiently represent and handle
- e.g. Complex numbers, Points in coordinates, all kinds of records (student records, police records, etc.)
- a large number of these types are compound types - mixture of **heteregenous (mixed) types**
  - e.g. student records may have integer for ID, string for names and addresses, float for GPA and grades, etc.
- the following figure shows some sample student records that a program may have to represent
- two records displayed in the figure have the same heteregeneous structure
  - we can represent these records and store them in memory using array of structure
- via **struct** and **class** constructs, C++ allows us to create any type of heteregeneous data records that we want to represent
- we do not include any library to use **struct** and **class** keywords
- **class** is a big topic that typically is covered in more details in **CS2** or in courses like *Object Oriented Programming*

1.3 Structures

- structrues are user-defined, compound and typically heterogenous types
- the following figure demonstrates student record represented using strcuture
- allows us to organize different types of data under one compound type
- each type of data is represented by its own name and is called a member of the structure
- makes it easier to manipulate and move the data record around using a single object or variable
- using structures is a two-step process
  - 1. define the new structure type
  - 2. declare objects using the new structrue type
- keyword **struct** short for structure is used to define structure type

- syntax to define structure:

```
struct structureName {
    type1 memberName1;
    type2 memberName2;
    type3 memberName3;
    type4 memberName4;
    //...
};
```

- notice semi-colon (;) after closing curley brace
- we don't initialize members; they are merely the blueprint (template) not actual variables just yet!
- syntax to declare objects of struct type

```
structureName objectName;
```

- exactly like declaring simple variables
- the 2nd step actually allocates all the memory required to store one record for some instance objectName
- the process of creating objects from struct type is called **instantiation**
- syntax to access members

```
objectName.memberName
```

- each member is used like a single variable; only difference is the way they're accessed
- member can be accessed only by it's instance (object) name
- compound variables that could have more than 1 value are typically called **objects**

1.4 Point structure definition

- a point in Cartesian coordinate (2-d geometry) is two numbers called coordinates
- there may be a large number of points on the plain, but each point is treated collectective as a single object
- e.g.(0, 0) indicates the origin, and (x, y) indicates the point x units from x-axis and y units from the y-axis

- how can we represent 2-d points in C++?
  - use structure!

[1]:
```
#include <iostream>
#include <string>
#include <cmath>

using namespace std;
```

[2]:
```
// define a point structure
struct Point {
    // can be declared as int x, y;
    int x; // member 1
    int y; // member 2
    // any other member?
    // parenthesis are common on all points and used only for representation
    // we don't need members for parenthesis
};
// do not initialize members as the memory is not allocated just yet!
```

1.5 Point objects

- recall Point structure is just the definition and doesn't actually store data
- need to declare Point objects to actaully store the data values (coordinates)
- we can also declare pointers to struct types
- syntax to declare struct objects and pointers is similar to declaring variables
  - afterall, struct is a user-defined type

`structName objectName;`

- objects created are automatic or stored in stack memory segment

[3]:
```
// declare/instantiate some point objects
Point pt1, pt2;
```

[4]:
```
// declare and initialize point objects
// using uniform initialization
// members are initialized in the order they're defined
Point origin = {0, 0};
```

[5]:
```
// explictly casting two values as Point type
pt1 = Point({2, 3});
```

[6]:
```
// implict coersion of two values as a Point type
pt2 = {3, 0};
```

[7]:
```
// declared a pointer to Point type and initialize with nullptr
Point * pt_ptr = nullptr;
```

[8]:
```
// assign value/address to pt_ptr
// recall pointers store memory addressess only!
pt_ptr = &pt1;
```

[9]:
```
// two addresses must be equal!
cout << pt_ptr << " == " << &pt1 << endl;
```

```
0x10bcf72d0 == 0x10bcf72d0
```

1.5.1 Dynamic objects

- memory needed for any struct objects can be allocated in heap memory segment
- the syntax to allocate dynamic objects is same as declaring dynamic variable covered in **Pointers** chapter

```
structName * ptrName = new structName();
```

[10]:
```
// instantiate a pointer object
Point * pt_ptr1 = new Point;
```

[11]:
```
// instantiate and initialize a pointer object
Point * pt_ptr2 = new Point({100, -200});
```

1.6 Point members

- each member of Point objects can be accessed using `.` (period or dot) - member access operator
- syntax:

```
object.member;
ptrObject->member;
```

- members are same as variables we can store and access data
- if a pointer object is used, -> (arrow/pointer) operator is used to access member

[12]:
```
// access members using . (member access) operator
cout << "origin = (" << origin.x << "," << origin.y << ")" << endl;
```

```
origin = (0,0)
```

[13]:
```
// assgin values to pt1 and pt2;
pt1.x = -3;
pt1.y = 1;
```

[14]:
```
// find the distance between pt1 and pt2
float dist;
```

[15]:
```
dist = sqrt(pow(pt1.x-pt2.x, 2) + pow(pt1.y-pt2.y, 2))
```

[15]:
```
6.08276f
```

[16]:
```
cout << "distance = " << dist << endl;
```

```
distance = 6.08276
```

[17]:
```
// accessing members using pointer variables
pt_ptr1->x = -3;
pt_ptr1->y = 1;
```

[18]:
```
// we get the same result as above
dist = sqrt(pow(pt_ptr1->x-pt2.x, 2) + pow(pt_ptr1->y-pt2.y, 2))
```

[18]: 6.08276f

1.6.1 visualize struct and objects in pythontutor.com

1.7 Template structures

- notice that Point class defined above uses **int** as type for x and y coordinates
- what if we had a coordinate system that used floating point values
  - we'd have to define another struct to represent Point using floating point values
- similar to template function, we can use **template type** in struct definition
  - acts as a placeholder for type that will be passed when the objects are intantiated
- templated struct helps create one generic struct definition that meets all type requirments for its members
- syntax to define template struct type:

```
template<class T1, class T2, ...>
struct structName {
    T1 member1;
    T2 member2;
    type member3;
    // more templated type or actual type members
};
```

- notice the syntax is same as the function template syntax
- template<…> construct let's you use 1 or more template type separated by comma
- syntax to instantiate objects of template struct types

```
structName<actualType1, actualType2, ...> objectName;
```

- actualType1 replaces T1, acttualType2 replaces T2, and so on…

1.7.1 templated rectangle type

- sides of rectangle may be of various types such as integer, or float or double, etc.
- we define templated rectangle type to account for those types

[19]:
```
// assuming both length and width of any rectangle will have the same type T
template<class T>
```

```
struct Rectangle {
    T length, width;
    // could use an array of T type
    // T sides[2];
    // length and width are better names than array
};
```

[20]:
```
// instantiate some objects of Rectangle types
Rectangle<int> r1;
Rectangle<float> r2;
```

[21]:
```
// instantiate and initialize rectangle objects
Rectangle<int> r3 = {10, 5};
```

[22]:
```
Rectangle<float> r4 = {8.5f, 5.5f};
```

[23]:
```
Rectangle<double> r5 = {100.999, 55.898};
```

1.8 Aggregate operations on struct objects

- for any type one has to wonder what operators work out of the box
  - e.g. for string, we could use +, =, comparison operators (>, ==, etc.)
- no aggregate operations such as input and output are allowed on struct objects as a whole
  - e.g. can't read into (cin) or print (cout) objects
  - it may not make sense to compare two objects (compare based on what members?)
- for most operations (except for assignmet), objects must be accessed one member at a time!
  - Note, there are ways to explictly overload aggregate operations by writing extra code
  - that is usually covered in CS2 or *Object Oriented Programming* courses

[24]:
```
// try cout; can't!!
// pt1 is an object of Point type
//cout << pt1;
// cout may be broken if you run this! so restart the kernel if you get error
```

```
input_line_35:4:6: error: invalid operands to binary

expression ('std::__1::ostream' (aka 'basic_ostream<char>') and 'Point')
cout << pt1;
~~~~ ^  ~~~

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:219:20:

note: candidate function not viable: no known conversion from
'Point' to 'const void *' for 1st argument; take
      the address of the argument with &
    basic_ostream& operator<<(const void* __p);
```

```
                  ^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/type_traits:4034:3:

note: candidate function not viable: no known conversion from
'std::__1::ostream' (aka 'basic_ostream<char>') to
      'std::byte' for 1st argument
  operator<< (byte  __lhs, _Integer __shift) noexcept
  ^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:195:20:

note: candidate function not viable: no known conversion from
'Point' to 'std::__1::basic_ostream<char>
      &(*)(std::__1::basic_ostream<char> &)' for 1st argument
    basic_ostream& operator<<(basic_ostream& (*__pf)(basic_ostream&))
                   ^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:199:20:

note: candidate function not viable: no known conversion from
'Point' to 'basic_ios<std::__1::basic_ostream<char,
      std::__1::char_traits<char> >::char_type, std::__1::basic_ostream<char,
std::__1::char_traits<char>
      >::traits_type> &(*)(basic_ios<std::__1::basic_ostream<char,
std::__1::char_traits<char>
      >::char_type, std::__1::basic_ostream<char, std::__1::char_traits<char>
>::traits_type> &)' (aka
      'basic_ios<char, std::__1::char_traits<char> > &(*)(basic_ios<char,
std::__1::char_traits<char> >
      &)') for 1st argument
    basic_ostream& operator<<(basic_ios<char_type, traits_type>&
                   ^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:204:20:

note: candidate function not viable: no known conversion from
'Point' to 'std::__1::ios_base
      &(*)(std::__1::ios_base &)' for 1st argument
    basic_ostream& operator<<(ios_base& (*__pf)(ios_base&))
                   ^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:207:20:

note: candidate function not viable: no known conversion from
'Point' to 'bool' for 1st argument
    basic_ostream& operator<<(bool __n);
                   ^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:208:20:

note: candidate function not viable: no known conversion from
'Point' to 'short' for 1st argument
```

```
    basic_ostream& operator<<(short __n);
                   ^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:209:20:
note: candidate function not viable: no known conversion from
'Point' to 'unsigned short' for 1st argument
    basic_ostream& operator<<(unsigned short __n);
                   ^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:210:20:
note: candidate function not viable: no known conversion from
'Point' to 'int' for 1st argument
    basic_ostream& operator<<(int __n);
                   ^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:211:20:
note: candidate function not viable: no known conversion from
'Point' to 'unsigned int' for 1st argument
    basic_ostream& operator<<(unsigned int __n);
                   ^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:212:20:
note: candidate function not viable: no known conversion from
'Point' to 'long' for 1st argument
    basic_ostream& operator<<(long __n);
                   ^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:213:20:
note: candidate function not viable: no known conversion from
'Point' to 'unsigned long' for 1st argument
    basic_ostream& operator<<(unsigned long __n);
                   ^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:214:20:
note: candidate function not viable: no known conversion from
'Point' to 'long long' for 1st argument
    basic_ostream& operator<<(long long __n);
                   ^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:215:20:
note: candidate function not viable: no known conversion from
'Point' to 'unsigned long long' for 1st argument
    basic_ostream& operator<<(unsigned long long __n);
                   ^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:216:20:
note: candidate function not viable: no known conversion from
```

```
'Point' to 'float' for 1st argument
    basic_ostream& operator<<(float __f);
                   ^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:217:20:

note: candidate function not viable: no known conversion from
'Point' to 'double' for 1st argument
    basic_ostream& operator<<(double __f);
                   ^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:218:20:

note: candidate function not viable: no known conversion from
'Point' to 'long double' for 1st argument
    basic_ostream& operator<<(long double __f);
                   ^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:220:20:

note: candidate function not viable: no known conversion from
'Point' to
      'basic_streambuf<std::__1::basic_ostream<char, std::__1::char_traits<char>
>::char_type,
      std::__1::basic_ostream<char, std::__1::char_traits<char> >::traits_type>
*' (aka
      'basic_streambuf<char, std::__1::char_traits<char> > *') for 1st
argument
    basic_ostream& operator<<(basic_streambuf<char_type, traits_type>* __sb);
                   ^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:223:20:

note: candidate function not viable: no known conversion from
'Point' to 'std::nullptr_t' (aka 'nullptr_t') for
      1st argument
    basic_ostream& operator<<(nullptr_t)
                   ^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:760:1:

note: candidate function not viable: no known conversion from
'Point' to 'char' for 2nd argument
operator<<(basic_ostream<_CharT, _Traits>& __os, char __cn)
^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:793:1:

note: candidate function not viable: no known conversion from
'Point' to 'char' for 2nd argument
operator<<(basic_ostream<char, _Traits>& __os, char __c)
```

```
^
/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:800:1:
note: candidate function not viable: no known conversion from
'Point' to 'signed char' for 2nd argument
operator<<(basic_ostream<char, _Traits>& __os, signed char __c)
^
/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:807:1:
note: candidate function not viable: no known conversion from
'Point' to 'unsigned char' for 2nd argument
operator<<(basic_ostream<char, _Traits>& __os, unsigned char __c)
^
/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:821:1:
note: candidate function not viable: no known conversion from
'Point' to 'const char *' for 2nd argument
operator<<(basic_ostream<_CharT, _Traits>& __os, const char* __strn)
^
/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:867:1:
note: candidate function not viable: no known conversion from
'Point' to 'const char *' for 2nd argument
operator<<(basic_ostream<char, _Traits>& __os, const char* __str)
^
/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:874:1:
note: candidate function not viable: no known conversion from
'Point' to 'const signed char *' for 2nd argument
operator<<(basic_ostream<char, _Traits>& __os, const signed char* __str)
^
/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:882:1:
note: candidate function not viable: no known conversion from
'Point' to 'const unsigned char *' for 2nd argument
operator<<(basic_ostream<char, _Traits>& __os, const unsigned char* __str)
^
/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:1066:1:
note: candidate function not viable: no known conversion from
'Point' to 'const std::__1::error_code' for 2nd
      argument
operator<<(basic_ostream<_CharT, _Traits>& __os, const error_code& __ec)
^
/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:753:1:
note: candidate template ignored: deduced conflicting types for
```

```
parameter '_CharT' ('char' vs. 'Point')
operator<<(basic_ostream<_CharT, _Traits>& __os, _CharT __c)
^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:1057:1:

note: candidate template ignored: could not match
'basic_string_view<type-parameter-0-0, type-parameter-0-1>'
      against 'Point'
operator<<(basic_ostream<_CharT, _Traits>& __os,
^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:1086:1:

note: candidate template ignored: could not match
'unique_ptr<type-parameter-0-2, type-parameter-0-3>' against
      'Point'
operator<<(basic_ostream<_CharT, _Traits>& __os, unique_ptr<_Yp, _Dp> const&
__p)
^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/iomanip:477:5:

note: candidate template ignored: could not match
'__iom_t10<type-parameter-0-0>' against 'Point'
    operator<<(basic_ostream<_Cp, _Traits>& __os, const __iom_t10<_Cp>& __x);
    ^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/iomanip:572:33:

note: candidate template ignored: could not match
'__quoted_output_proxy<type-parameter-0-0, type-parameter-0-2,
      type-parameter-0-1>' against 'Point'
basic_ostream<_CharT, _Traits>& operator<<(
                                ^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/iomanip:592:33:

note: candidate template ignored: could not match
'__quoted_proxy<type-parameter-0-0, type-parameter-0-1,
      type-parameter-0-2>' against 'Point'
basic_ostream<_CharT, _Traits>& operator<<(
                                ^

/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:1039:1:

note: candidate template ignored: requirement
'!is_lvalue_reference<basic_ostream<char> &>::value' was not
      satisfied [with _Stream = std::__1::basic_ostream<char> &, _Tp =
Point]
operator<<(_Stream&& __os, const _Tp& __x)
```

```
^
/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:814:1:
note: candidate template ignored: could not match 'const _CharT
*' against 'Point'
operator<<(basic_ostream<_CharT, _Traits>& __os, const _CharT* __str)
^
/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:1093:1:
note: candidate template ignored: could not match
'bitset<_Size>' against 'Point'
operator<<(basic_ostream<_CharT, _Traits>& __os, const bitset<_Size>& __x)
^
/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/valarray:4165:1:
note: candidate template ignored: substitution failure [with
_Expr1 = std::__1::basic_ostream<char>, _Expr2 =
      Point]: no type named 'value_type' in 'std::__1::basic_ostream<char>'
operator<<(const _Expr1& __x, const _Expr2& __y)
^
/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/valarray:4180:1:
note: candidate template ignored: substitution failure [with
_Expr = std::__1::basic_ostream<char>]: no type
      named 'value_type' in 'std::__1::basic_ostream<char>'
operator<<(const _Expr& __x, const typename _Expr::value_type& __y)
^
/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/valarray:4196:1:
note: candidate template ignored: substitution failure [with
_Expr = Point]: no type named 'value_type' in
      'Point'
operator<<(const typename _Expr::value_type& __x, const _Expr& __y)
^
/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:1049:1:
note: candidate template ignored: could not match
'basic_string<type-parameter-0-0, type-parameter-0-1,
      type-parameter-0-2>' against 'Point'
operator<<(basic_ostream<_CharT, _Traits>& __os,
^
/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/ostream:1074:1:
note: candidate template ignored: could not match
'shared_ptr<type-parameter-0-2>' against 'Point'
operator<<(basic_ostream<_CharT, _Traits>& __os, shared_ptr<_Yp> const& __p)
```

```
^
/Users/rbasnet/anaconda3/envs/cpp/include/c++/v1/iomanip:362:1:
note: candidate template ignored: could not match
'__iom_t8<type-parameter-0-2>' against 'Point'
operator<<(basic_ostream<_CharT, _Traits>& __os, const __iom_t8<_MoneyT>& __x)
^
```

```
Interpreter Error:
```

[24]:
```
// read in/store data one member at a time
char ch;
Point pt3;
```

[26]:
```
cout << "Enter a point in (x, y) format: ";
cin >> ch >> pt3.x >> ch >> pt3.y >> ch;
// ch is just a place variable for unnecessary character to read and ignore
```

```
Enter a point in (x, y) format: (1, 2)
```

[26]:
```
@0x10bcaad70
```

[27]:
```
// print the point in right format; accessing one member at a time
cout << "pt3 = (" << pt3.x << ", " << pt3.y << ")";
```

```
pt3 = (1, 2)
```

1.8.1 aggregate copy (=) is allowed

- one struct object can be copied into another out of the box
- object is copied member by member from source to destination

[28]:
```
Point pt4 = pt3;
```

1.9 Passing struct objects to functions

- struct objects can be passed to functions both by value and by reference

1.9.1 pass by value

- struct objects can be copied into another same type of struct objects using (=) assignment operator
- this allows us to pass struct to functions by value (by copying the data)

[29]:
```
// passing some constant Point
void printPoint(const Point pt) {
    cout << "(" << pt.x << ", " << pt.y << ")";
}
```

[30]:
```
printPoint(pt4);
```

```
(1, 2)
```

1.9.2 pass by reference

- any data type can be explictly pass by reference in C++

[31]:
```
void getPoint(Point & pt) {
    cout << "Enter a point in (x, y) format: ";
    cin >> ch >> pt.x >> ch >> pt.y >> ch;
    // Note: when using terminal, after the last character ) is read \n is left␣
 ↪behind
    // getline() will fail!
    // good idea to read \n whitespace and ignore it!
}
```

[32]:
```
Point pt5;
```

[34]:
```
getPoint(pt5);
```

```
Enter a point in (x, y) format: (8, 0)
```

[35]:
```
printPoint(pt5);
```

```
(8, 0)
```

[36]:
```
// function finds the distance between two points
// sqrt( (x1-x2)^2 + (y1-y2)^2 )
float distance(const Point & p1, const Point & p2) {
    return sqrt(pow(p1.x-p2.x, 2) + pow(p1.y-p2.y, 2));
}
```

[37]:
```
cout << "distance between ";
printPoint(pt4);
cout << " and ";
printPoint(pt5);
cout << " = " << distance(pt4, pt5);
```

```
distance between (1, 2) and (8, 0) = 7.28011
```

1.10 Returning struct from functions

- as the (=) assignment works on structs, functions can return struct types

[38]:
```
// function returns Point type object
Point getPoint() {
    Point pt;
    cout << "Enter a point in (x, y) format: ";
    cin >> ch >> pt.x >> ch >> pt.y >> ch;
    return pt;
}
```

[39]:
```
// assign the returned object from getPoint() to pt6 object
Point pt6 = getPoint();
```

```
Enter a point in (x, y) format: (4, 4)
```

[40]:
```
printPoint(pt6);
```

```
(4, 4)
```

1.11 Array/vectors of structs

- if more than one similar records/structs need to be stored
  - we can use array or vector of struct type
- let's say we need to store a bunch of points in memory
  - array of points is a natural choice

[41]:
```
// declare and initialize array
Point points[] = {{1, 2}, {3, 4}, {6, 7}, {-1, -1}, {0, 0}};
```

[42]:
```
// declare array of points
Point points1[2];
```

[43]:
```
// accessing point element in array
printPoint(points[0]);
```

```
(1, 2)
```

[44]:
```
// accessing point element's member in array
cout << "first point's x = " << points[0].x << endl;
```

```
first point's x = 1
```

[45]:
```
// assiging values to array
points1[0] = getPoint();
```

```
Enter a point in (x, y) format: (10, 5)
```

[46]:
```
points1[1] = getPoint();
```

```
Enter a point in (x, y) format: (-4, -10)
```

1.11.1 vectors of struct type

- vector like array can be used to store user-defined data types using **struct**

[54]:
```
// declare and initialize vector of Point
vector<Point> point_vector = {{0, 0}, {1, 1}, {2, 2}};
```

[48]:
```
// create vector of RectangleType
vector<Rectangle<int> > rects;
```

[49]:
```
// add r1 rectangle object to rects vector
rects.push_back(r1);
```

[51]:
```
// can't add Rectangle r2 because its type is float
rects.push_back(r3);
```

[52]:
```
// declare and initialize rectangles vector with two rectangles
vector<Rectangle<float> > rectangles = {{10, 5}, {8.5, 2.6}};
```

[55]:
```
// calculate area of first rectangle stored in rectangles vector
cout << "area = " << rectangles[0].length*rectangles[1].width << endl;
```

```
area = 26
```

[56]:
```
// traversing vectors
// auto also works on user-defined type
for(auto rect: rectangles) {
    cout << "rectangle info - length x width: " << rect.length << " x " << rect.
 ↪width << endl;
}
```

```
rectangle info - length x width: 10 x 5
rectangle info - length x width: 8.5 x 2.6
```

[ ]:
```
// same as above
for(RectangleType rect: rectangles) {
    cout << "rectangle info - length x width: " << rect.length << " x " << rect.
 ↪width << endl;
}
```

[57]:
```
// using index
for(int i=0; i<rectangles.size(); i++) {
    cout << "rectangle area: "
        << rectangles[i].length << "x"
        << rectangles[i].width << " = "
        << rectangles[i].length*rectangles[i].width << endl;
}
```

```
rectangle area: 10x5 = 50
rectangle area: 8.5x2.6 = 22.1
```

1.12 Array/vector in struct

- array or vector of any type can be used as a member of a struct
- if there are several members of same types that don't need their own names, we can use an array member
- having each member their own identifier makes program more readable and struct intuitive to use

[ ]:
```
// let's define a structure to store student record
struct Student {
    string firstName;
    char MI;
    string lastName;
    float test_scores[3]; // each test don't have a unique name
    string pri_contact_fName;
    char pri_contact_MI;
    string pri_contact_lName;
    bool semester_finished[2]; // semesters though have names Freshman Fall,␣
 ↪etc.; we can use 1st, 2nd etc.
};
```

[4]:
```
// declaration of st1
Student st1;
```

[5]:
```
st1.firstName = "John";
```

[6]:
```
// accessing an array member
// NOTE: array can be accessed one element at a time
st1.test_scores[0] = 100;
st1.test_scores[1] = 95;
```

[7]:
```
// accessing another array member
st1.semester_finished[0] = true;
st1.semester_finished[1] = false;
```

[8]:
```
// instantiate and initialize
// Note the order of values and how each member is initialized based on its type
Student st2 = {"Jane", 'A', "Smith", {0, 0, 0}, "Jim", 'J', "Smith", {false,␣
 ↪false}};
```

1.13 Struct in another struct

- any struct type can be used as a member type in another struct
- in Student structure above, firstName, MI and lastNames can be repeated for various names
  - student name, primary contact, secondary contact, father's name, mother's name, etc.

- we can convert the repeating group of members into its own struct type

[2]:
```
// most people have three names
struct NameType {
    string firstName;
    char MI;
    string lastName;
};
```

[3]:
```
// let's redifine Student type with NameType
struct StudentType {
    NameType name;
    float test_scores[3];
    NameType primary_contact;
    bool semester_finished[2];
};
// Notice how shorter the StudentType has become
// We've not used created other name type, but just imagine each name is just␣
 ↪one member!
// makes the StudentType concise yet readable and intuitive
```

[4]:
```
// instantiate objects
StudentType st3;
```

[5]:
```
// assign values to name member
// "name" is a member of st3 object but it itself is a struct type object
// keep drilling down until we come to the actual member name that stores the␣
 ↪data
st3.name.firstName = "David";
st3.name.MI = 'A';
st3.name.lastName = "Johnson";
```

[9]:
```
// shorter way to assign to a struct type object
st3.name = {"Dave", 'A', "Johnson"};
```

[6]:
```
// create an array of student records
StudentType students[2];
```

[15]:
```
students[0] = st3;
```

[16]:
```
// access member of array and member of struct
students[0].semester_finished[0] = true;
```

1.14 Exercises

1. Write a program that computes distance between two points in Cartesian coordinates.
   - use struct to represent Point

- prompt user to enter two points
- use as many function(s) as possible
- write at least 3 test cases for each computing functions
- program continues to run until user wants to quit
- most of the part is done in Jupyter Notebook demo

2. Write a program to compute area and circumference of a circle using struct.
   - use struct to represent Circle
   - prompt user to enter radius of a circle
   - use as many function(s) as possible
   - write at least 3 test cases for each computing functions
   - program continues to run until user wants to quit
3. Write a program to compute area and perimeter of a rectangle using struct.
   - use struct to represent Rectangle
   - prompt user to enter length and width of a rectangle
   - use as many function(s) as possible
   - write at least 3 test cases for each computing functions
   - program continues to run until user wants to quit
4. Write a program to compute area and perimeter of a triangle given 3 sides.
   - use struct to represent Triangle
   - prompt user to enter 3 sides of a triangle
   - use as many function(s) as possible
   - write at least 3 test cases for each computing functions
   - program continues to run until user wants to quit

[1]:
```
// Sample solution for #4
// using incremental development
// using functions as possible to break the problem
#include <iostream>
#include <cmath>
#include <string>
#include <cassert>
#include <sstream>
#include <iomanip>

using namespace std;
```

[2]:
```
// use struct to represent Triangle
// could be a templated struct
struct Triangle {
    float side1, side2, side3;
    // can be an array
    // float sides[3];
};
```

[3]:
```
// function to check if 3 sides form a triangle
bool validTriangle(float s1, float s2, float s3) {
    // sum of every pair must be greater than the third
```

```
    return (s1+s2 > s3 && (s2+s3 > s1) && (s1+s3 > s2))? true: false;
}
```

[4]:
```
void test_validTriangle() {
    assert(validTriangle(2, 3, 4) == true);
    assert(validTriangle(1, 2, 3) == false);
    assert(validTriangle(4, 5, 10) == false);
    cerr << "all test cases passed for validTriangle()\n";
}
```

[5]:
```
test_validTriangle()
```

```
all test cases passed for validTriangle()
```

[6]:
```
// function prompts user to enter 3 sides of a triangle
// creates and returns a triangle
Triangle getTriangle() {
    float s1, s2, s3;
    // input validation
    do {
        cout << "Enter three sides of a triangle separated by space: ";
        cin >> s1 >> s2 >> s3;
        // check if three sides form a triangle
        if (!validTriangle(s1, s2, s3))
            cout << "3 sides do not form a traingle.\n"
                << "Sum of any 2 sides must be greater than the third!\n";
        else
            break;
    } while(true);
    return Triangle({s1, s2, s3});
}
```

[7]:
```
// let's manually test getTriangle
Triangle t1;
```

[8]:
```
t1 = getTriangle();
```

```
Enter three sides of a triangle separated by space:
1 2 3
3 sides do not form a traingle.
Sum of any 2 sides must be greater than the third!
Enter three sides of a triangle separated by space: 3 4 5
```

[11]:
```
float trianglePerimeter(const Triangle & t) {
    return t.side1 + t.side2 + t.side3;
}
```

```
input_line_19:1:7: error: redefinition of
'trianglePerimeter'
float trianglePerimeter(const Triangle & t) {
      ^
input_line_17:1:7: note: previous definition is
here
float trianglePerimeter(const Triangle & t) {
      ^
```

Interpreter Error:

[12]:
```
// write 3 test cases for trianglePerimeter
void test_trianglePerimeter() {
    assert(trianglePerimeter(Triangle({2, 3, 4})) == 9);
    assert(trianglePerimeter(Triangle({3, 4, 5})) == 12);
    assert(trianglePerimeter(Triangle({2.5, 3.5, 4.5})) == 10.5);
    cerr << "all test cases passed for trianglePerimeter()\n";
}
```

```
input_line_20:2:6: error: redefinition of
'test_trianglePerimeter'
void test_trianglePerimeter() {
     ^
input_line_18:2:6: note: previous definition is
here
void test_trianglePerimeter() {
     ^
```

Interpreter Error:

[13]:
```
test_trianglePerimeter();
```

```
all test cases passed for trianglePerimeter()
```

[14]:
```
// function to compute area of a triangle
float triangleArea(const Triangle & t) {
    // use heron's formula: https://www.mathsisfun.com/geometry/herons-formula.
 ↪html
    float s = trianglePerimeter(t)/2;
```

```
    return sqrt(s*(s-t.side1)*(s-t.side2)*(s-t.side3));
}
```

[15]:
```
// wrapper function to test if two floating numbers are equal upto precision␣
↪decimal points
void assertAlmostEqual(float value1, float value2, int precision) {
    ostringstream oss;
    // create output string stream with precision for floating-point values
    oss << fixed << setprecision(precision) << value1 << " " << value2;
    // create input string stream from output string stream
    istringstream iss(oss.str());
    float v1, v2;
    // extract the values as float
    iss >> v1 >> v2;
    assert(v1 == v2);
}
```

[16]:
```
// write 3 test cases for triangleArea
void test_triangleArea() {
    assert(triangleArea(Triangle({3, 4, 5})) == 6.0);
    float area = triangleArea({2, 4, 5}); // coersion of 3 values into Triangle␣
↪type
    assertAlmostEqual(area, 3.799671038392666, 4); // accuracy upto 4 decimal␣
↪points
    assertAlmostEqual(triangleArea({3, 4, 6}), 5.3326822, 4);
    cerr << "all test cases passed for triangleArea()\n";
}
```

[17]:
```
test_triangleArea();
```

```
all test cases passed for triangleArea()
```

[18]:
```
// function to calculate and print the result on triangle
void printResult(const Triangle & t) {
    cout << "Triangle info: \n"
         << "3 sides length: " << t.side1 << " " << t.side2 << " " << t.side3
         << "\narea: " << triangleArea(t)
         << "\nperimeter: " << trianglePerimeter(t);

}
```

[19]:
```
// complete program
void program() {
    Triangle t;
    string cont;
    do {
        t = getTriangle();
```

```
        printResult(t);
        cout << "\nWant to enter another triangle? [yes|y]: ";
        cin >> cont;
        if (cont == "yes" || cont == "y") continue;
        else break;
    }while(true);
    cout << "Good bye...";
}
```

[20]:
```
program();
```

```
Enter three sides of a triangle separated by space: 1 2 3
3 sides do not form a traingle.
Sum of any 2 sides must be greater than the third!
Enter three sides of a triangle separated by space: 4 5 6
Triangle info:
3 sides length: 4 5 6
area: 9.92157
perimeter: 15
Want to enter another triangle? [yes|y]: yes
Enter three sides of a triangle separated by space: 4 5 6
Triangle info:
3 sides length: 4 5 6
area: 9.92157
perimeter: 15
Want to enter another triangle? [yes|y]: no
Good bye…
```

see complete sample solution for exercise 4 at demo_programs/Ch12/triangle.cpp

5. A Grade Book:
    - Write a C++ program that let's professors keep track of students grades with the following requirements:
    - program must use struct to keep track of students grades (at least 3 grades)
    - program prompts user to enter students information as many as they wish
    - program calculates avearge grade and the letter grade (A-F) based on the average grade
    - program sorts the student records based on grade in non-increasing order (highest to lowest)

1.15 Kattis problems

- struct is not a strict rquirement to solve Kattis problems
- struct is generally used when the problems can be better solved using your own type

1.16 Summary

- this chapter covered a new concept of creating user-defined type using struct
- saw many examples of struct types and objects instantiated with those types

- learned that array can be a member of struct
- learned that a larger number of records (struct type) can be stored in an array
- learned about out-of-the-box aggregate operations on struct objects
  - assignment (=) is the only available aggregate operation!
- learned how to pass struct objects to functions and return from them as well
- exercises and a sample solutions using incremental development technique

[ ]: