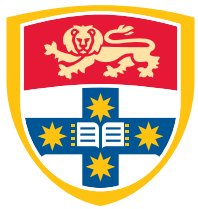
Introduction to Programming (Adv)

School of Computer Science, University of Sydney



Copyright Warning

Lecture 6: Defining and using types

New data types

Data types

Programming languages predefine data types based on the goals of the language

Memory is the important concept to understand when defining the new data type

Behaviour of the data type *may* also be defined

We look at two kinds of data type definitions to help us understand memory

Aggregate types

New data types can only be built from a combination of existing types

Lowest common denominator - char, int, float...

Called aggregate or composite types

Aggregate

Consider a supermarket item.

It has a 6 character barcode, fractional price and integer stock.

Define the memory only

```
1 struct super_item {
2     char barcode[6];
3     float price;
4     int stock;
5 };
```

Aggregate (cont.)

Defines BOTH the memory and initial values

```
1 class super_item:
2     def __init__(self):
3         self.barcode = ''
4         self.price = 0.0
5         self.stock = 0
```

Using values from outside

```
1 class super_item:
2     def __init__(self, barcode, price, stock):
3         self.barcode = barcode
4         self.price = price
5         self.stock = stock
```

What are the data types?

Aggregate initialisation

```
1  struct super_item {
2      char barcode[6];
3      float price;
4      int stock;
5  };
6
7  int main() {
8      struct super_item yoghurt;
9
10     yoghurt.price = 5.6;
11     yoghurt.stock = 14;
12     strncpy(yoghurt.barcode, "394729", 6);
13
14     printf("%s %.2f %d\n", yoghurt.barcode, yoghurt.price,
           yoghurt.stock);
15     return 0;
16 }
```

Aggregate initialisation (cont.)

```
1  struct super_item {
2      char barcode[6];
3      float price;
4      int stock;
5  };
6
7  int main() {
8      struct super_item yoghurt = { "394729", 5.6, 14 };
9      printf("%s %.2f %d\n", yoghurt.barcode, yoghurt.price,
           yoghurt.stock);
10     return 0;
11 }
```

Aggregate initialisation (cont.)

```
1  struct super_item yoghurt = { "394729", 5.6, 14 };
2  struct super_item milk = { "121212", 3.0, 78 };
3  struct super_item *ptr;
4
5  ptr = &yoghurt;
6  (*ptr).price = 5.2; // price change
7  printf("%s %d %d\n", (*ptr).barcode, (*ptr).price, (*ptr).
       stock);
8
9  ptr = &milk;
10 (*ptr).stock -= 2; // less 2 items
11 printf("%s %d %d\n", (*ptr).barcode, (*ptr).price, (*ptr).
       stock);
```

Aggregate initialisation (cont.)

```
1  struct super_item items[3];
2
3  for (int i = 0; i < 3; ++i)
4      items[i].price = 0.0;
5      items[i].stock = 0;
6      strncpy(items[i].barcode, "000000", 6);
```

```
1  struct super_item items[3] = {
2          { "000000", 0.0, 0},
3          { "000000", 0.0, 0},
4          { "000000", 0.0, 0}
5      };
```

Aggregate initialisation (cont.)

```
1 struct super_item items[3];
2
3 for (int i = 0; i < 5; ++i)
4     items[i].price = 0.0;
5     items[i].stock = 0;
6     strncpy(items[i].barcode, "000000", 6);
```

Operating on the type

```
1  struct super_item {
2      char barcode[6];
3      float price;
4      int stock;
5  };
6
7  void update_price(struct super_item *item,
8                    float new_price) {
9      (*item).price = new_price;
10 }
```

Operating on the type (cont.)

```
1  class super_item:
2      def __init__(self):
3          self.barcode = ''
4          self.price = 0.0
5          self.stock = 0
6
7      def update_price(self, new_price):
8          self.price = new_price
```

Union type

Share the same memory

```
1  union vector{
2      struct { int x, y; } position;
3      struct { int r, g, b; } colour;
4  };
5
6  int main() {
7      union vector v;
8
9      v.position.x = 1;
10     v.position.y = 2;
11
12     v.colour.r = 7;
13     v.colour.g = 8;
14     v.colour.b = 9;
15
16     return 0;
17 }
```

Union type (cont.)

Why the union type?

Problem: supermarket items have many differences. struct is excessive.

```
1  struct super_item {
2      char barcode[6];
3      float price;
4      int stock;
5      int dual_front_stock;
6      int dual_back_stock;
7      int cold_stock;
8      float cold_temp_min;
9      float cold_temp_max;
10     int hot_stock;
11     float hot_temp_min;
12     float hot_temp_max;
13     int hot_max_time;
14 };
```

Why the union type? (cont.)

```
1  struct super_item1 {
2      char barcode[6];
3      float price;
4      int stock;
5  }; // basic
6  struct super_item2 {
7      char barcode[6];
8      float price;
9      int cold_stock;
10     float cold_temp_min;
11     float cold_temp_max;
12 }; // cold specific
13 struct super_item3 {
14     char barcode[6];
15     float price;
16     int front_stock;
17     int back_stock;
18 }; // dual stock specific
```

Why the union type? (cont.)

Combinations of `super_item` 1 & 2, or 2 & 3, ... require more struct definitions!

Having many new types breaks existing functions to operate on

Solution: Share regions of memory that are NOT COMMON between items

Why the union type? (cont.)

```
1  enum stock_type { SINGLE, DUAL };
2  enum item_specific { HOT, COLD, NONE };
3
4  struct super_item {
5
6      // common for all items
7      char barcode[6];
8      float price;
9
10     enum stock_type stock_type; // single, dual
11     union {
12         int total;
13         struct {
14             int front_stock;
15             int back_stock;
16         } dual;
17     } stock;
18
```

Why the union type? (cont.)

```
19
20      enum item_specific item_specific; // hot, cold, none
21      union {
22          struct {
23              float temp_min;
24              float temp_max;
25          } cold;
26          struct {
27              float temp_min;
28              float temp_max;
29              int max_time;
30          } hot;
31          // ... more
32      } specific;
33  };
```

Why the union type? (cont.)

```
1  int main() {
2      struct super_item s;
3      strncpy(s.barcode, "391010", 6);
4      s.price = 2.40;
5
6      s.stock_type = DUAL;
7      s.stock.dual.front_stock = 13;
8      s.stock.dual.back_stock = 47;
9
10     s.item_specific = HOT;
11     s.specific.hot.temp_min = 28;
12     s.specific.hot.temp_max = 35;
13     s.specific.hot.max_time = 60*15;
14     return 0;
15 }
```

Summary

Composing a new data type uses existing types

Memory layout of the data type

Behaviour of the type

struct - stores multiple data types, each with their own memory

union - stores multiple data types, sharing the same memory

Pointers and self are related