SFU CMPT-212 2008-1 Topic: Compound Types

Ján Maňuch

E-mail: jmanuch@sfu.ca

Friday 25th January, 2008

Compound types

- arrays
- structures
- unions
- pointers

Arrays

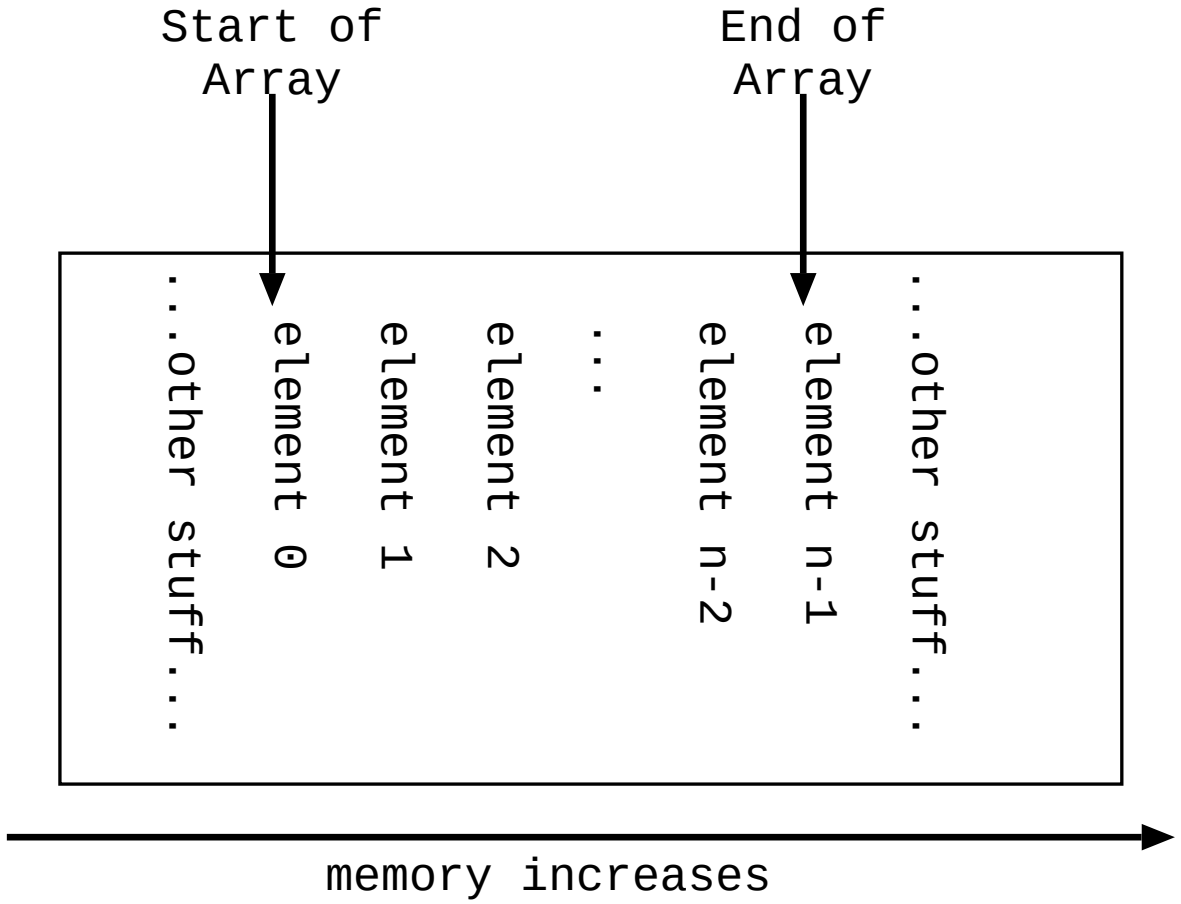
- a compound type representing a direct concatenation of elements of the same type.
- declaration of an array variable:
  `type_name array_name[array_size];`
  Examples:

```
1 int a[128];
2 bool b[10];
3 long long c[1024], d[10];
```

- accessing element of the array: `array_name[index]`
  Note: C++ arrays start with index 0 (like in "Java", unlike "Pascal").
  Example: the elements of an array `int x[3]` are `x[0]`, `x[1]`, `x[2]`
  Example: array.cpp

Arrays in memory

```
1 const int n=10;
2 double Array[n];
```



Remarks

- By ISO C++ standard 98, the size of the array should be *fixed* during the compilation time, i.e., `array_size` can be a constant, a `const` value, or a constant expression.

  Note. GNU gcc compiler allows variable size arrays, but it's not a standard;

  for details, see "variable len arrays"

- Array declarations can be combined with *scalar* variable declarations. *Example:*

```
1 int e = 0, f, g[120], h[2000];
```

Array initialization

- After declaring an array, the values of elements of the array are *undefined*!
- Arrays can be initialized using blocks of comma delimited values during declaration.

  Example:

```
1 int a[2] = { 101, 102 };
2 int b[5] = { 2, 4 }; // initialized to { 2,4,0,0,0 }
```

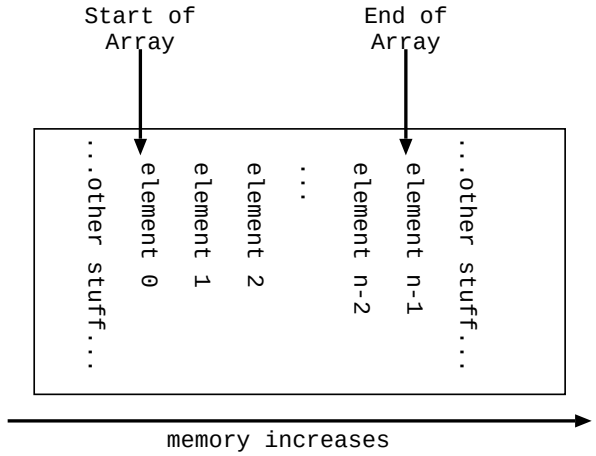
- If the array is initialized, you do not need to specify the size of the array. The size will be automatically computed to fit the content.

  Example:

```
1 int flexarray[] = { 0, 1, 2, 3, 4 };
2 // array of size 5
```

Remarks

- C/C++ arrays are *unsafe*, in the sense that there is no `'bounds overflow'` error if exceeding the size of the array.
  Example: unsafearrays.cpp



  - You *cannot* copy elements of one array to another with a single assignment:

```
1 int a[3]={2,4,8},b[3];
2 b=a; // compiler error
```

  - The operator `sizeof` can be used to calculate the memory used by an entire array (not by a cell).

    Example:

    `type onearray[size];`

    `sizeof(onearray)` will be equal to `size` $\times$ `sizeof(type)`

Multidimensional Arrays

- Multidimensional arrays can be described as *arrays of arrays*, i.e. arrays whose elements are arrays.

Examples:

```
1  short chessboard[8][8];
2  int maxtemps[4][3];
3  float acube[12][12][12];
```

`maxtemp` is an array with 4 elements, each being an array of 3 elements.

Initializing a multiarray:

```
1  int maxtemps[4][3]={
2    {1,2,3}, {11,12,13},
3    {101,102,103}, {4,5,6}
4  };
```

Example: multiarrays.cpp, multiarraysize.cpp (`sizeof`)

Arrays of characters

- arrays of characters (`char`) are used to represent C-style *strings*; string starts at position 0 and ends when the element containing the character with ASCII code '0' (null) is encountered

  Example:

```
1 char name[6]={'J','a','n','o','\0','M'};
2 // represents string "Jano"
3 cout <<name; // prints 'Jano'
```

- There is a simpler way how to initialize a string: a *string constant*:

```
1 char name[7]="Jano";
2 char name[]="Jano";  // array of size 5
```

Example: cmpt.cpp

Remarks

- The size of an array needed to accomodate a given string is *one more than the length of the string*. (The terminating `null` character must also be counted).

  Example:

```
1   char a[] = "hello";
2   cout << sizeof(a);   // prints 6
3   char a[5] = "hello"; // compiler error
```

Example: strings.cpp (based on [Prata])

Analyzing strings.cpp

- new header file `cstring`
  contains function `strlen()` returning the length of the string (without the null character)
- `cin >> name1;` takes only the first word (until space or end of line is encountered) of the input string
- you can use `getline()` or `get()` to read all until end of line:

```
1 const maxsize=20;
2 char name[maxsize];
3 cin.getline(name,maxsize);
```

Example: strings2.cpp

Structures

Structures are complex types, that contain named specified *data members*.

```
1 struct structure_name?
2  {
3    [type member_name;]+
4  } variable_name*;
```

where

‘?’ indicates “at most once”;

‘*’ indicates “zero or more times.”;

‘+’ indicates “one or more times.”

Structures

Example:

```
1   struct StudentStructure212
2   {
3     char name[30];
4     int assignment1,assignment2,assignment3,assignment4;
5     int midterm,final;
6     float mark;
7   } astudent, bstudent;
8
9   StudentStructure212 cstudent = {
10    "John", 100, 100, 60, 75, 89, 79, 85.4f
11  }, allstudents[32];
```

Structures

Data members of "structure" variables are accessed using the *membership operator* (`.`).

```
1    astudent.mark=79.3f;
```

Example: student.cpp (shows how to initialize a structure)

Memberwise assignment — when assigning one structure to another of the same type

Example: student2.cpp

(in reality, the whole block of memory occupied by structure is copied)

Unions

Unions are *polymorphic* complex data types. Their data members must be in *either / or* relationships.

Example: a product can be uniquely identified by *either* a product number *or* a product name

```
1 union ProductId {
2   long number;
3   char name[11];
4 } watermelon, shoe;
5
6 strcpy(watermelon.name, "watermelon");
7 shoe.number = 89123;
```

Example: union.cpp

Sample questions

- What's wrong with the following code?

```
1 int a=10;
2 int b[a];
```

- How would you declare an array of `int` of size 100 and initialize all its elements to 0's using just one statement?
- What is the value of `b` after executing the following code:

```
1 char a[]="cmpt";
2 int b=sizeof(a);
```

- In the following code, for each of the lines (except Line 1) specify the output printed to screen.

```
1 int a[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
2 cout <<a[2][1]<<endl;
3 cout <<a[3][0]<<endl;
4 cout <<a[2][4]<<endl;
5 cout <<a[0][6]<<endl;
6 cout <<a[-1][6]<<endl;
```

- How would you use `struct` type to copy content of an array with a single assignment?

- Consider the following two arrays:

```
1 int field2D[10][7];
2 int field[70];
```

  Assume that you want to use `field` to simulate two-dimensional array `field2D`. How would you access the element in `field` corresponding to the element `field2D[i][j]`?

- Consider the following declarations:

```
1 struct S {
2   int a;
3   int b;
4 } s;
5 union U {
6   int a;
7   int b;
8 } u;
```

and the following piece of code:

```
1 s.a = 1;
2 u.a = 1;
```

What are the values of `s.b` and `u.b`?

Pointers

- each byte in the memory of the computer has its *address* (usually it is a 4-byte integer, written in hexadecimal form)
- a variable containing such an address is called a *pointer*
- in C++, there are many types of pointers:
  - each type has its own pointer
  - a pointer for certain type "should" point to (contain an address of) a place in the memory where a value of this type is stored
- the *address operator* "`&`" can be applied on any variable to get the address (pointer) where the variable is stored

```
1 int a;
2 cout <<&a; \\ print the address of "a"
```

- the *dereferencing operator* "`*`" can be applied on any pointer to access the value pointed to by the pointer

Declaration of pointers

```
type * pointer_to_type;
```

```
type * pointer_to_type;
```
Example:

```
1 int * pa;    // a pointer pointing to an int value
2 int a=2;
3 pa = &a;     // "pa" points the location of "a"
4 cout << *pa; // prints the value at location "pa" (2)
```

Example: pointer.cpp [Prata]

Comments:

- the spaces around "`*`" in declaration are optional
- `int *pa;` — emphasizes that `*pa` is of type `int`
- `int* pa;` — emphasizes that `int*` is a type "pointer to `int`"
- **beware:** `int * p, q;` declares `p` to be a pointer to `int` and `q` to be an `int`

Comments:

- A constant that represents the value of a pointer that *points to nothing* is denoted as `NULL`.
  `NULL` is not part of the C++ language, but is included in standard libraries (e.g. `iostream`).

- *Pointers are unsafe*: the compiler **cannot** guarantee that a pointer is pointing to a meaningful location.

```
1 double *p; // points to random address
2 *p=10.12;  // modifies sizeof(double) bytes
3            // at this random address
```

- C++ is more restrictive about types when using pointers
  Example: typecheck.cpp

- Pointers and typecasting, used together, can be a very powerful, yet very dangerous combination.

  Example:

```
1  char c = 69;
2  long *p = NULL;
3
4  p = 1; // compiler error: cannot assign an int to a pointer
5
6  p = (long*)(1); // typecasting int to pointer to long
7  // the pointer points to some weird address
8  // reading the value will likely generate a runtime error
9
10 p = &c; // compiler error: incompatible pointer types
11
12 p = (long*)(&c); // no compiler errors, but accessing more bytes
13
14 cout << p << " : " << *p << "\n";
```

  Example: pointerunsafe.cpp

Pointer arithmetic

Some arithmetic operations on pointers make sense.

Example: consider

```
int i, *P, *Q;
```

- A pointer to the **i-th** element of the *pointed type*, **after** `*P`, in contiguous memory (result: a pointer):
  `P+i`
- A pointer to the **i-th** element of the *pointed type*, **before** `*P`, in contiguous memory (result: a pointer):
  `P-i`
- The *distance* between two pointers, i.e. the number of values of the *pointed type* that can fit between the two pointed values (result: an integer value):
  `P-Q`

Arrays and Pointers

- Any array variable is [almost] equivalent to a pointer pointing to the beginning in memory of the array.

Example: hello.cpp

`*(a + i)` is exactly the same thing as `a[i]`

Example: the following declarations are equivalent:

```
1 char a[] = "hello";
2 // an array declaration, including initialization
3 char *a  = "hello";
4 // a pointer declaration,
5 // initialized to the address of a constant
```

Arrays and Pointers — differences

- Array variables are slightly better behaved (during compilation) than pointers.

  Example: pintersandarrays.cpp

  - `sizeof` – array variable knows how many elements it contains
  - the address where the array variable points cannot be changed

Remark:

- `cout` (and also `cin`) is smart again (`char*` variable printed as string) — use typecasting to print the address
  Note: when reading to `char *ps` make sure `ps` points to allocated space!

Arrays of pointers

Pointer variables can form arrays, just as basic variables.

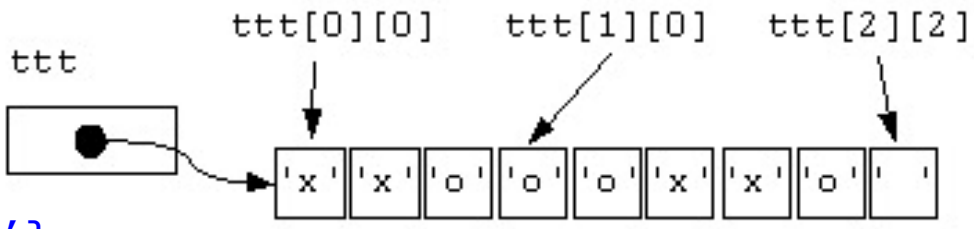
Example:

```
1 int *a[10];
2 int *b[] = { NULL, NULL, NULL, NULL };
3 char *days[] = {"Monday", "Tuesday", "Wednesday", "Thursd
```

Example: arraysofpointers.cpp

Note. *Multidimensional arrays **are not** pointers to pointers.*

Multidimensional arrays are arrays.



Example:

```
1 char ttt[3][3] = {{'x', 'x', 'o'},
2                   {'o', 'o', 'x'},
3                   {'x', 'o', ' '}};
```

Example: multiarrays.cpp

Parameters to main(...)

Console applications written in C/C++ accept *command-line parameters* that are strings (separated by spaces), passed as parameters to the `main` function.

Syntax:

```
int main (int argn, char *argv[])
```

- `int argn` : the Number of Command Line Parameters.
- `char* argv[]` : an array of strings that are the command-line parameters.

Example: parameters.cpp

Sample questions

- Which of the following lines will generate a compiler error?

```
1 int a;
2 int* p,q;
3 p=10;
4 p=&a;
5 q=10;
6 q=&a;
```

- Why is it ok to assign `short` to `long`, but it's not ok to assign `short*` to `long*`?

- Consider the following two declarations:

```
1   int a[10];
2   int *b=a;
```

What are the two differences between `a` and `b`?

- Assume you declared `int *p`. What are the differences among `p++`, `*p++` and `(*p)++`?
- What is wrong with the following code?

```
1 char *pCity;
2 cout <<"Enter a city: ";
3 cin >> pCity;
```