505 22240 / ESOE 2012 Data Structures: Lecture 2
Pointers, Arrays, Loops, and Functions

◎Pointers

・A pointer is a variable that holds the value of a variable's address in memory.

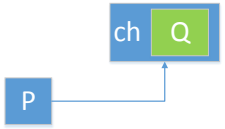
・Given a type T, the type `T*` denotes a pointer to a variable of type T. e.g. `int*` denotes a pointer to an integer.

・The address-of operator, `&`, returns the address of an object in memory. e.g. `X` is an integer variable and `&X` is the address of X in memory.

・Accessing an object's value from its address is called "dereferencing", which is done using the `*` operator (return the contents of a given address).

・e.g. `q` be a pointer to an integer and set `q=&X`, we could access X's value with `*q`.

・Examples:




```
char ch = 'Q';

char* P = &ch;      // P holds the address of ch
cout << *P;         // outputs the character 'Q'
ch ='Z';            // ch now holds 'Z'
cout << *P;         // outputs the character 'Z'
*P= 'X';            //ch now holds 'X'
cout << ch;         //outputs the character 'X'
```

・Null pointer: a pointer value that points to nothing.

・The `*` operator binds with the variable name, not with the type name.

・e.g. `int* x, y, z;` `// same as int* x; int y; int z;`

⇨ One pointer variable x and two integer variables y and z.

◎Arrays

・An array is a collection of elements of the same type.

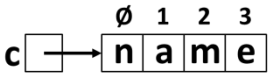
・An object consisting of a numbered list of variables that are identically typed entities.

・Given any type T and a constant N, a variable of type T[N] holds an array of N elements, each of type T (index from 0 to N-1).

・Once declared, it is not possible to increase the number of elements in an array.

・Examples:

```
double f[5];        // array of 5 doubles: f[0],…,f[4]
int m[10];          // array of 10 ints: m[0],…,m[9]
f[4] = 2.5;
m[2] = 4;
cout << f[m[2]];        // outputs f[4], which is 2.5
```




```
char c[4];              // declaration of a new array
c[0] = 'n';
c[1] = 'a';
c[2] = 'm';
c[3] = 'e';
int length = sizeof(c) / sizeof(c[0]);
// calculate the length of the array
```

・Initializing an array with curly braces "{"and "}".

```
int a[ ]={10, 11, 12 ,13}; // declares and initializes a[4]
bool b[ ]={false, true};  // declares and initializes b[2]
char c[ ]={'c', 'a', 't'}; // declares and initializes c[3]
```

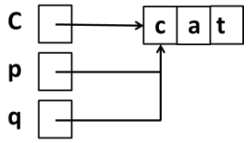
・Declare an array of pointers to integers:

`int* r[17]` declares an array r consisting of 17 pointers to objects of type int.

`*r[16]` is the value of the integer pointed to by the last element of this array.

◎Pointers and Arrays

· The name of an array is equivalent to a pointer to the array's initial element and vice versa, e.g.,




```
char c[ ] = {'c', 'a', 't'};
char* p = c;       // p points to c[0]
char* q = &c[0];   // q also points to c[0]
cout << c[2] << p[2] << q[2];      // outputs "ttt"
```

◎C-Style Structures

· A structure is a data type that groups other data types together into a single, compound data type.

· Examples:

```
enum MealType {NO_PREF, REGULAR, LOW-FAT, VEGETARIAN};
//define a new type called Passenger
struct Passenger {
  string name;              // passenger name
  MealType mealPref;        // meal preference
  bool isFreqFlyer;    // in the frequent flyer program?
  string freqFlyerNo; // passenger's freq. flyer number
};
//declare and initialize a variable name "pass"
Passenger pass = {"John Smith", VEGETARIAN, true, "293145"};
```

· Member selection operator (.) with the form `struct_name.member`.

```
pass.name = "Peter Jackson";   // change name
pass.mealPref = REGULAR;       // change meal preference
```

· If P1 and P2 are of the same type, Passenger, then P2 = P1 copies the elements of P1 to P2.

◎Pointers, Dynamic Memory, and the "new" Operator

· Free store (heap): a large block of memory for creating dynamic objects ⇨ dynamic memory.

· The operator `new` dynamically allocates memory for an object of a given type and returns a pointer to this object.

· The operator `->` is used to access members of an object with pointer.

```
(*pointer_name).member = pointer_name->member
```

· Example:

```
Passenger *p;
//…
p = new Passenger;     // p points to the new Passenger
p->name = "Diana";     // set the structure members
p->mealPref = REGULAR;
p->isFreqFlyer = false;
p->freqFlyerNo = "NONE";
//…
delete p;          // destroy the object p points to
```

· The `delete` operator only to objects allocated through `new`.

```
delete ObjPointer
delete [ ] ArrayPointer
```

· Example:

```
char* buffer = new char[500];
// allocate a buffer of 500 chars
buffer[3] = 'a';       // accessing elements using [ ]
delete [ ] buffer;     // delete the buffer
```

★Memory Leaks: inaccessible objects in dynamic memory.

⇨If an object is allocated with **new**, it should eventually deallocated with **delete**.

◎References

· A reference is simply an alternative name for an object.

· Given a type T, the notation T& indicates a reference to an object of type T.

· Example:

```
string author = "Samuel Clemens";
string& penName = author;
// penName is an alias for author
penName = "Mark Twain";    // author = "Mark Twain"
cout << author;            // outputs "Mark Twain"
```

· A reference must refer to an actual variable.

· References are often used for passing function arguments and returning results from functions.

◎Constants and Typedef

· Use all capital letters when naming constants:

```
const double PI = 3.14159265;
const int CUT_OFF[ ] = {90, 80, 70, 60};
const int N_DAYS = 7;
const int N_HOURS = 24*N_DAYS;   // constant expression
int counter[N_HOURS];          // an array of 168 ints
```

· An alias declared for an existing data type with a **typedef** declaration:

```
typedef char* BufferPtr;
// type BufferPtr is a pointer to char
typedef double Coordinate;
// type Coordinate is a double
BufferPtr p;           // p is a pointer to char
Coordinate x, y;       // x and y are of type double
```

◎Local and Global Scopes

```
const int Cat = 1;     // global Cat
int main( ) {
  const int Cat = 2;           // this Cat is local to main
  cout << Cat;             // outputs 2 (local Cat)
  return EXIT_SUCCESS;
}
int dog = Cat;         // dog = 1 (from the global Cat)
```

§ **Expressions**

- An expression combines variables and literals with operators to create new values.

◎Member Selection and Indexing

class_name.member class / structure member selection

`pointer->member`	class / structure member selection
`array[exp]`	array subscripting

◎Arithmetic Operators

`exp + exp`	addition
`exp - exp`	subtraction
`exp * exp`	multiplication
`exp / exp`	division
`exp % exp`	remainder

◎Increment and Decrement Operators

`var++`	post increment
`var--`	post decrement
`++var`	pre increment
`--var`	pre decrement

· Example:

```
int a[ ] = {0, 1, 2, 3};
int i = 2;
int j = i++;            // j=2 and now i=3
int k = --i;            // now i=2 and k=2
cout << a[k++];         // a[2]=2 is output; now k=3
```

◎Relational and Logical Operators (true or false)

`exp < exp`	less than
`exp > exp`	greater than
`exp <= exp`	less than or equal to
`exp >= exp`	greater than or equal to
`exp == exp`	equal to
`exp != exp`	not equal to
`!exp`	logical not
`exp && exp`	logical and
`exp || exp`	logical or

◎Bitwise Operator

`~exp`	bitwise complement
`exp & exp`	bitwise and
`exp ^ exp`	bitwise exclusive - or
`exp | exp`	bitwise or
`exp1 << exp2`	shift exp1 left by exp2 bits
`exp1 >> exp2`	shift exp1 right by exp2 bits

◎Assignment Operators (`=`)

- `n += 2` means `n = n + 2`

◎Other Operators

· Scope resolution operator (`::`): access nested structure members.

`class_name::member`	class scope resolution
`namespace_name::member`	namespace resolution
`bool_exp ? true_exp : false_exp`	conditional expression

· Example: `smaller = (x < y ? x : y)` `// smaller = min(x, y)`

◎Changing Types through Casting

★Let exp be some expression, and let T be a type.

・C-style cast : `(T)exp`

・Functional-style cast : `T(exp)`

```
int cat = 14;
double dog = (double)cat;
// traditional C-style cast
double pig = double(cat);
// C++ functional cast
int i1 = 18;
int i2 = 16;
double dv1 = i1/i2;                    // dv1 = 1.0
double dv2 = double(i1)/double(i2);    // dv2 = 1.125
double dv3 = double(i1/i2);            // dv3 = 1.0
```

★Static casting

・Static casting is used when a conversion is made between two related types, e.g., numbers to numbers or pointers to pointers.

```
static_cast<desired_type>(expression)
```

・Example: (truncation)

```
double d1 = 3.2;
double d2 = 3.9999;
int i1 = static_cast<int>(d1);         // i1 = 3
int i2 = static_cast<int>(d2);         // i2 = 3
```

§ Control Flow

◎**If** Statement

```
if (condition)
  true_statement
else if (condition)
  else_if_statement
…
else
  else_statement
```

・Example:

```
if (snowLevel < 2) {
  // do these if snow level is less than 2
  goToClass( );
  comeHome( );
}
else if (snowLevel < 5)
  // if level is at least 2 but less than 5
  haveSnowballFight( );
else if (snowLevel < 10)
  // if level is at least 5 but less than 10
  goSkiing();
else
  // if snow level is 10 or more
  stayAtHome( );
```

◎**Switch** Statement (integral type or enumeration)

```
char command;
cin >> command;          // input command character
```

```
switch(command) {          // switch based on command value
    case 'I':              // if (command == 'I')
        editInsert( );
        break;
    case 'D':              // else if (command == 'D')
        editDelete( );
        break;
    case 'R';              // else if (command == 'R')
        editReplace( );
        break;
    default:               // else
        cout << "Unrecognized command";
        break;
}
```

§ Loops

◎While and Do-While Loops

```
while (condition)
    loop_body_statement
```

· Example:

```
int a[100];
//...
int i = 0;
int sum = 0;
while (i < 100 && a[i] >= 0) {
    sum += a[i++];
}
```

· e.g. isPrime

```
bool isPrime(int n) {
    int divisor = 2;
    while (divisor < n) {
        if (n % divisor == 0) {
            return false;
        }
        divisor++;
    }
    return true;
}
```

⇨ loop body

· If n ≤ 2, the loop body won't iterate even once.

★Do-While

```
do
    loop_body_statement
while (condition)
```

· Example:

```
int counter = 5;
int factorial = 1;
do {
    factorial *= counter--;
} while (counter > 0);
std::cout << "factorial of 5 is " << factorial << std::endl;
```

◎**For** Loop: Equivalent to "while" Loops

```
· for (initialization; test; update) {
     statements;
  }

· initialization
   while (test) {
     statements;
     update;
  }
```

· Example:

```
const int NUM_ELEMENTS = 100;
double b[NUM_ELEMENTS];
for (int i = 0; i < NUM_ELEMENTS; i++) {
     if (b[i] > 0)
          cout << b[i] << '\n';
}
```

· e.g. modify the isPrime

```
bool isPrime(int n) {
     for (int divisor = 2; divisor < n; divisor++) {
          if (n % divisor == 0) {
               return false;
          }
     }
     return true;
}
```

◎Loop Bounds

· Print all primes in range 2…n

```
void printPrimes(int n) {
     int i;
     for (i = 2; i < n; i++) {
     // ERROR: condition should be i <= n
          if (isPrime(i)) {
               cout << " " << i;
          }
     }
}
```

★Common loop bounds

```
     for (int i = 0; i < n; i++)
     for (int i = 1; i <= n; i++)
```

◎Break and Continue Statements

· A break statement is used to "break" out of a loop or switch statement.

· The continue statement can only be inside loops (for, while, and do while). It causes the execution to skip to the end of the loop, ready to start a new iteration.

· Example:

```
int a[100];
//…
int sum = 0;
for (int i = 0; i < 100; i++) {
     if (a[i]<0) break;
     sum += a[i];
}
```

◎Primes Revisited: Sieve of Eratosthenes

```
   2  3  4  5  6  7  8  9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
          ⋮
```

```
void printPrimes(int n) {
    bool prime[n+1];
    int i;
    for (i = 2; i <= n; i++) {
        prime[i] = true;
    }
    for (int divisor=2; divisor*divisor <= n; divisor++) {
        if (prime[divisor]) {
            for (i = 2*divisor; i <= n; i = i + divisor) {
                prime[i] = false;
            }
        }
    }
    for (i = 2; i <= n; i++) {
        if (prime[i]) {
            cout << " " << i;
        }
    }
}
```

§ Functions

- A function is a chunk of code that can be called to perform some well-defined task.

```
Return type Function name(Argument list) {
    Function body: collection of C++ statements
}
```

· Example:

```
bool evenSum(int a[ ], int n);  // function declaration:
appear in every file that invokes the function
int main( ) {
  int list[ ] = {4, 2, 7, 8, 5, 1};
  bool result = evenSum(list, 6);    // invoke the function
  if (result)    cout << "the sum is even \n";
  else           cout << "the sum is odd \n";
  return EXIT_SUCCESS;
}
```

```
//function definition: appear only once
bool evenSum(int a[ ], int n) {
  int sum = 0;
  for (int i = 0; i < n; i++)
    sum += a[i];
  return (sum % 2) == 0;     // returns true if sum is even
}
```

◎Argument Passing

· Arguments in C++ programs are passed by value (default).

· Passing the argument by reference:

```
void f(int value, int& ref) {
// arguments with one value and one reference
  value++;       // no effect on the actual argument
  ref++;         // modifies the actual argument
  cout << value << endl;     // outputs 2
  cout << ref << endl;       // outputs 6
}

int main( ) {
  int cat = 1;
  int dog = 5;
  f(cat, dog);       // pass cat by value, dog by ref
  cout << cat << endl;       // outputs 1
  cout << dog << endl;       // outputs 6
  return EXIT_SUCCESS;
}
```

· Constant reference: to avoid changing the data being referenced.

```
void someFunction(const Passenger& pass) {
  pass.name = "New Name";
  // ILLEGALL! pass is declared const
}
```

◎Overloading and Inlining

- Overloading: two or more functions or operators have the same name, but whose effect depends on the types of their actual arguments.

★ Function overloading: same name, different argument lists.

```
// print an integer
void print(int x)
  {cout << x;}
// print a Passenger
void print(const Passenger& pass) {
  cout << pass.name << "  " << pass.mealPref;
  if (pass.isFreqFlyer)
    cout << "  " << pass.freqFlyerNo;
}
```

★ Operator overloading: operators such as +, *, +=, ==, and << …

```
bool operator==(const Passenger& x, const Passenger& y) {
    return x.name == y.name
    && x.mealPref == y.mealPref
    && x.isFreqFlyer == y.isFreqFlyer
    && x.freqFlyerNo == y.freqFlyerNo;
}
```

★ In-line Functions: very short functions

```
inline int min(int x, int y) {return (x < y ? x : y);}
```