Practical C/C++ programming
Part II

Feng Chen
IT Analyst 3
Louisiana State University
Quick review of Part I

- Introduction to C and C++ language
- Basic syntax and grammar
- Data types, constants and variables:
  - Basic types (integer, float, void)
  - Derived types (arrays)
- Operators
  - Arithmetic
  - Logical
  - Relational
  - Misc (sizeof, ",", ternary, etc)
- Control Flow
- Functions
- Input/Output control
Things to be covered today

- Pointers in C/C++
  - Use in functions
  - Use in arrays
  - Use in dynamic allocation
- User defined type
  - struct
- Introduction to C++
  - Changes from C to C++
  - C++ class and objects
- Introduction to common C++ libraries
Pointers

- Pointers are a very important part of the C programming language. They are used in many ways, such as:
  - Array operations (e.g., while parsing strings)
  - Dynamic memory allocation
  - Sending function arguments by reference
  - Generic access to several similar variables
  - Malloc data structures of all kinds, especially trees and linked lists
  - Efficient, by-reference “copies” of arrays and structures, especially as function parameters

- Necessary to understand memory and address...and the C programming language.
What is a pointer?

- A pointer is essentially a **variable** whose value is the address of another variable.
- Since it is a variable, it must be declared before use.
- Pointer “points” to a specific part of the memory.
- How to define pointers?
  
  ```
  type *pointer_var_name;
  ```

- Examples
  
  ```
  int   *i_ptr;   /* pointer to an integer */
  double *d_ptr; /* pointer to a double */
  float  *f_ptr; /* pointer to a float */
  char   *ch_ptr; /* pointer to a character */
  int   **p_ptr; /* pointer to an integer pointer */
  ```
Pointer rules

- There are two prefix unary operators to work with pointers.
  & /* "address of" operator */
  * /* "dereferencing" operator */

- Use ampersand “&” in front of a variable to access it's address, this can be stored in a pointer variable.

- Use asterisk “*” in front of a pointer you will access the value at the memory address pointed to (dereference the pointer).

- Examples:

```c
int a = 6;
int *p;
/* point p to a */
p = &a;
/* dereference pointer p */
*p = 10;
```

<table>
<thead>
<tr>
<th>var_name</th>
<th>var_address</th>
<th>var_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0x22aac4</td>
<td>6</td>
</tr>
<tr>
<td>p</td>
<td>0x22aac0</td>
<td>0x22aac4</td>
</tr>
</tbody>
</table>
/* pointer_rules.c */
#include <stdio.h>

int main() {
    int a = 6, b = 10;
    int *p;
    printf("Initial values: the value of a is %d, value of b is %d
", a, b);
    printf("the address of a is: %p, address of b is: %p
", &a, &b);
    p = &a; /* point p to a */
    printf("after p = &a: 
");
    printf("the value of p is %p, value at that address is %d
", p, *p);
    p = &b; /* point p to b */
    printf("after p = &b: 
");
    printf("the value of p is %p, value at that address is %d
", p, *p);
    /* dereference pointer p */
    *p = 6, p = &a, *p = 10;
    printf("after dereferencing the pointer: 
");
    printf("the value of a is %d, value of b is %d
", a, b);
    return 0;
}
Never dereference an uninitialized pointer!

- In order to dereference the pointer, pointer must have a valid value (address).
- What is the problem for the following code?
  ```c
  int *ptr;
  *ptr = 3;
  ```
- Again, you will have **undefined behavior** at runtime, you are operating on unknown memory space.
- Typically error: “Segmentation fault”, possible illegal memory operation
- **Always initialize your variables before use!**

<table>
<thead>
<tr>
<th>var_name</th>
<th>var_address</th>
<th>var_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>0x22aac0</td>
<td>0xXXXX</td>
</tr>
<tr>
<td></td>
<td>0xXXXX</td>
<td>3</td>
</tr>
</tbody>
</table>
NULL pointer

- Memory address 0 has special significance, if a pointer contains the null (zero) value, it is assumed to point to nothing, defined as NULL in C.
- Set the pointer to NULL if you do not have exact address to assign to your pointer.
- A pointer that is assigned NULL is called a null pointer.
  ```c
  /* set the pointer to NULL 0 */
  int  *ptr = NULL;
  ```
- Before using a pointer, ensure that it is not equal to NULL:
  ```c
  if (ptr != NULL) {
    /* make use of pointer1 */
    /* ... */
  }
  ```
As we have learned from Part I, in C, arguments are passed by value to functions: changes of the parameters in functions do **not** change the parameters in the calling functions.

Take a look at the below example, what are the values of a and b after we called swap(a, b);

```c
/* this is the main calling function */
int main() {
    int a = 2;
    int b = 3;
    printf("Before:  a = %d and  b = %d\n", a, b);
    swap(a, b);
    printf("After:  a = %d and  b = %d\n", a, b);
}
/* this is function, pass by value */
void swap(int p1, int p2) {
    int t;
    t = p2, p2 = p1, p1 = t;
    printf("Swap: a (p1) = %d and  b(p2) = %d\n", p1, p2);
}
```
Pointers and Functions (2)

- The values of a and b do not change after calling swap(a,b)
- **Pass by value means the called functions' parameter will be a copy of the callers' passed argument.** The value of the caller and called functions will be the same, but the identity (the variable) is different - caller and called function each has its own copy of parameters
- Solution at this point? Using pointers

```c
/* pass by pointer */
void swap_by_reference(int *p1, int *p2) {
    int t;
    t = *p2, *p2 = *p1, *p1 = t;
    printf("Swap: a (p1) = %d and b(p2) = %d\n", *p1, *p2);
}

/* call by-address function */
swap_by_reference( &a, &b );
```

2/26/2014
Pointers and Arrays (1)

- The most frequent use of pointers in C is for walking efficiently along arrays.
- **Remember, array name is the first element address of the array (it is a constant)**

```c
int *p=NULL;  /* define an integer pointer p*/
/* array name represents the address of the 0th element of the array */
int a[5]={1,2,3,4,5};
/* for 1d array, below 2 statements are equivalent */
p = &a[0];    /* point p to the 1st array element (a[0])'s address */
p = a;        /* point p to the 1st array element (a[0])'s address */
*(p+1);      /* access a[1] value */
*(p+i);      /* access a[i] value */
p = a+2;      /* p is now pointing at a[2] */
p++;         /* p is now at a[3] */
p--;         /* p is now back at a[2] */
```

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
</table>

2/26/2014

Practical C/C++ programming II
Pointers and Arrays (2)

- Recall 2D array structure: combination of 1D arrays
  
  ```
  int a[2][2]={{1,2},{3,4}};
  ```

- The 2D array contains 2 1D arrays: array `a[0]` and array `a[1]`

- `a[0]` is the address of `a[0][0]`, i.e:
  - `a[0]` is the address of `a[0][0]`
  - `a[1]` is the address of `a[1][0]`

- **Array a** is then actually an **address array** composed of `a[0]`, `a[1]`, i.e. `a` is the address of `a[0]`

<table>
<thead>
<tr>
<th>col 0</th>
<th>col 1</th>
<th>array a[0]</th>
<th>array a[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>row 0</td>
<td>a[0][0]=1</td>
<td>a[0][1]=2</td>
<td>a[1][0]=3</td>
</tr>
<tr>
<td>row 1</td>
<td>a[1][0]=3</td>
<td>a[1][1]=4</td>
<td></td>
</tr>
</tbody>
</table>

array a:

- `a` is the address of `a[0]`
- `a[0]` is the address of `a[0][0]`
- `a[1]` is the address of `a[1][0]`
Walk through array with pointer

```c
#include <stdio.h>
const int MAX = 3;
int main () {
    int a_i[] = {10, 20, 30};
double a_f[] = {0.5, 1.5, 2.5};
    int i;
    int *i_ptr;
    double *f_ptr;
    /* let us have array address in pointer */
i_ptr = a_i;
f_ptr = a_f;
    /* use the ++ operator to move to next location */
    for (i=0; i<MAX; i++, i_ptr++, f_ptr++) {
        printf("adr a_i[%d] = %8p\t", i, i_ptr);
        printf("adr a_f[%d] = %8p\n", i, f_ptr);
        printf("val a_i[%d] = %8d\t", i, *i_ptr);
        printf("val a_f[%d] = %8.2f\n", i, *f_ptr);
    }
    return 0;
}
```
Dynamic memory allocation using pointers

- For situations that the size of an array is unknown, we must use pointers to dynamically manage storage space.
- C provides several functions for memory allocation and management.
- Include `<stdlib.h>` header file to use these functions.
- Function prototype:

```c
/* This function allocates a block of num bytes of memory and return a pointer to the beginning of the block. */
void *malloc(int num);

/* This function release a block of memory block specified by address. */
void free(void *address);
```
Example of 1D dynamic array

/* dynamic_1d_array.c */
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    int n;
    int* i_array; /* define the integer pointer */
    int j;
    /* find out how many integers are required */
    printf("Input the number of elements in the array:\n");
    scanf("%d",&n);
    /* allocate memory space for the array */
    i_array = (int*)malloc(n*sizeof(int));
    /* output the array */
    for (j=0;j<n;j++) {
        i_array[j]=j; /* use the pointer to walk along the array */
        printf("%d ",i_array[j]);
    }
    printf("\n");
    free((void*)i_array); /* free memory after use*/
    return 0;
}
How to make dynamic 2D array?

- Use dynamic 2D array in Exercise 3 (refer to /*dynamic_2d_array.c*/)
  
  - Hint:
    
    /* First malloc a 1D array of pointer to pointers, then for each address, malloc a 1D array for value storage: */
    
    int** array;
    array=(int**)malloc(nrows*sizeof(int*));
    for (i=0; i<nrows; i++)
        array[i]=(int*)malloc(ncols*sizeof(int));
    /* DO NOT forget to free your memory space */
    for (i=0; i<nrows; i++)
        free((void*)array[i]);
    free((void*)array);

  - Question:
    
    - What is the difference between the dynamic 2D array generated using the above method and the static 2D one defined using the method in Part 1 slide (page 45)? (Hint: check whether the memory for the dynamic 2D array is contiguous by print the address of the pointer array)
    
    - Any solutions to the above method? (This method will be important when being used in MPI (Message Passing Interface) function calls)
Structures

- User-defined type in C: `struct`, union and enum
- A C `struct` is an aggregate of elements of (nearly) arbitrary types.
- Structures are the basic foundation for objects and classes in C++.
- Structures are used for:
  - Passing multiple arguments in and out of functions through a single parameter
  - Data structures such as linked lists, binary trees, graph, and more

Syntax for defining structure:

```c
/* syntax for defining structure */
struct [structure tag] /* tag is optional */
{
    member definition;
    member definition;
    ...
    member definition;
} [one or more structure variables];
```
How to use struct

- Example of defining a “Point” struct
  ```c
  /* define a structure “Point” */
  struct Point {
    int index;
    char tag;
    double x;
    double y;
  };
  ```

- Define the struct Point type variables:
  ```c
  /* define two struct Point variables */
  struct Point p1, p2, p3;
  ```

- Here is how we access the struct Point type variables, using the “.”:
  ```c
  p1.index=0; /* access members of p1 with dot “.” operator */
  p1.tag = 'a';
  p1.x = 0.0;
  p1.y = 0.0;
  p3 = p1; /* assign struct variable p1 to variable p3 */
  ```
Using typedef to define new variables

- C provides a keyword called *typedef* to name a new variable type (note that typedef does not create new types).

```c
typedef existing_type new_type_name;
```

- Use *typedef* with struct in the previous example:

```c
typedef struct Point point;
typedef struct Point { /* alternative way to define Point*/
    int index;
    char tag;
    double x;
    double y;
} Point;
```

- *typedef* can also used to give alias to existing variable types:

```c
typedef double real; /* typedef float real; easy switch between precisions*/
```

- Use the newly defined type to define your variables, e.g.:

```c
real x; /* x is actually double defined above */
Point p1, p2, p3; /* p1, p2 and p3 are struct Point */
```
Pointer to struct

- Define pointers to structures in the same way as pointer to any other basic variables:
  ```c
  Point *ptr_p; /* define a pointer to Point */
  ```

- Use the pointer to point to the actual struct variable by: store the address of a structure variable in the above defined pointer variable with the address “&” operator:
  ```c
  ptr_p = &p3; /* point ptr_p to struct p3 */
  ```

- To access struct members with pointer, use the “->” operator
  ```c
  printf("tag=%c\n", ptr_p->tag); /* access the struct member through pointer */
  ```

- Alternatively, use the dereference operator “*” and the “.” operator
  ```c
  printf("tag=%c\n", (*ptr_p).tag); /* access the struct member through dereference operator */
  ```

- The “->” operator will be largely used in the class and object operations in C++
/* struct_example.c */
#include <stdio.h>

typedef double real;
/* typedef float real; easy switch between single and double precisions */

typedef struct Point {
  int index;
  char tag;
  real x;
  real y;
} Point;

void print_point(struct Point point);

int main() {
  /* define two struct Point variables */
  /* struct Point p1, p2; */
  Point p1, p2, p3;
  /* assign values to struct members of p1 */
  p1.index=0;
  p1.tag = 'a';
  p1.x = 0.0;
  p1.y = 0.0;
struct Example - Point struct (2)

25 /* assign values to struct members of p2 */
26 p2.index=1, p2.tag = 'b', p2.x = 1.0, p2.y = 1.0;
27 p3 = p1; /* assign struct var p1 to var p3 */
28 /* output p1 and p2 */
29 print_point(p1);
30 print_point(p2);
31 print_point(p3);
32 }
33
34 void print_point(struct Point point)
35 {
36     printf( "\npoint %c:\n", point.tag);
37     printf( "\nindex : %d\n", point.index);
38     printf( "\nx = %7.2lf\n", point.x);
39     printf( "\ny = %7.2lf\n", point.y);
40     printf( "\n" );
41 }

From C to C++

- C++ can be considered as a superset of C
- Some minor C++ features over C
  - You can use “//” to type a comments
  - To use standard C libraries: `using namespace std;`
  - Input from the keyboard and output to the screen can be performed through `cout` (insertion operator) and `cin` (extraction operator)
  - Variables can be declared anywhere inside the code (e.g. C++ allows you to declare a variable to be local to a loop)
  - Can use reference for a variable instead of pointer
  - Memory manipulation: `new` and `delete`

- Major difference: C is function-driven while C++ is object-driven. C is procedure oriented while C++ is **object oriented**.
- Will touch these features in the next section.
- To compile a C++ program, change the compiler name to `g++` using the GNU compiler:
  ```
  $ g++ cpp_features.cpp
  ```
Minor C++ features over C

```cpp
#include <iostream>

// use standard libraries
using namespace std;

// we are using C++ style comments
int main()
{
    int n = 2*3; // Simple declaration of n
    int *a = new int[n]; //use "new" to manage storage
    // C++ style output
    cout << "Hello world with C++" << endl;
    for (int i = 0; i < n ; i++)
    {
        a[i]=i;
        // we are using C++ cout for output
        cout << "a[" << i << "] = " " << a[i] << endl;
    }
    delete[] a; // free the memory space we used
    return 0;
}
```
References in C++

- C++ references allow you to create an alias for the variable which allows you to treat the reference exactly as though it were the original variable.

- Declaring a variable as a reference by appending an ampersand “&” to the type name, reference must be initialized at declaration:

  ```cpp
  int& rx = x; // declare a reference for x
  ```

- Example using C++ reference as function parameters (see `ref.cpp`):

  ```cpp
  int main() {
    int x,y=4;
    int& rx = x; // declare a reference for x
    rx = 3; // rx is now a reference to x so this sets x to 33
    cout << "before: x=" << x << " y=" << y << endl;
    swap(x,y);
    cout << "after: x=" << x << " y=" << y << endl;
  }
  ```

  ```cpp
  void swap (int& a, int& b) {// using reference instead of pointers
    int t;
    t=a,a=b,b=t;
  }
  ```
Major migration – Class and Object in C++

- Definition of class
  - A class is a user-defined type. It is an expanded concept of user-defined type struct.

- Definition of object
  - An object is an instance of a class.

- In terms of variables, a class would be the variable type, and an object would be the variable.

- In C++, Classes are defined using either keyword `class` or keyword `struct`, with the following syntax:

```cpp
// syntax for defining a class
class class_name {
    accessSpecifier_1: // private, public or protected
    member1;           // list of class members
    accessSpecifier_2:
    member2;
    ...
} [object_names];  // object_names is an optional list of this class
```
More on C++ class definition

- class_name is a valid identifier for the class
- object_names is an optional list of names for objects of this class.
- The body of the declaration can contain members, which can either be data or function declarations, and optionally access specifiers.
- An access specifier is one of the following three keywords:
  
  ```cpp
  private: // accessible only from within class or their "friends"
  public:   // The members declared as public are accessible from outside the class through an object of the class
  protected: // accessible from outside the class BUT only in a class derived (derived class) from it.
  ```

- By default, all members of a class is `private` unless access specifier is used.
- The definition is very similar to plain data struct except that they can also include `functions (methods) with access specifier`. 
Class example: Point class

Below is an example rewrite the Point struct to class:

```cpp
class Point {
    // define a class Point
private:
    // list of private members
    int index; // index of the point
    char tag; // name of the point
    real x; // x coordinate, real: typedef double real;
    real y; // y coordinate

public:
    // use this function to set the private members
    void set_values(int, char, real, real);
    // use this function to output the private members
    void print_values();
};

// define the "set_values" method
void Point::set_values(int idx, char tg, real xc, real yc) {
    index=idx, tag=tg, x=xc, y=yc;
}

// define the "print_values" method
void Point::print_values() {
    cout << "point " " tag " " index = " " index
    << ", x = " " x " " y = " " y " endl;
}
```
Some explanation of the Point class

- **private** members of Point: `index`, `tag`, `x`, `y` cannot be accessed from outside the Point class:
  - they have private access
  - they can only be accessed from within other members of that same class.

- **public** members of Point can be accessed as normal functions via the dot operator "." between object name and member name.

- The implementation of the member functions can be either inside or outside the class definition. In the previous slide, the member function is defined outside the class definition.

- The scope operator "::", for the function definition is used to specify that the function being defined is a member of the class Point and not a regular (non-member) function:

  ```cpp
  // define the "set_values" method using scope operator "::"
  void Point::set_values(int idx, char tg, real xc, real yc) {
    index=idx, tag=tg, x=xc, y=yc; // overuse of comma operator :-)
  }
  ```
Use class to define objects

- To declare objects of a class, use exactly the same type of declaration as declaring variables of basic types.
- Following statements declare two objects of class Point, just the same as we define basic type variables:
  ```
  Point p1, p2; // define two object of Point
  ```
- Then the objects p1 and p2 access their member functions:
  ```
  p1.set_values(0,'a',0,0); // object p1 use set_values method
  p2.set_values(1,'b',1,1); // object p2 use set_values method
  p1.print_values();  // object p1 use print_values method
  p2.print_values();  // object p2 use print_values method
  ```
- We cannot directly access the private members of p1 and p2:
  ```
  double x1= p1.x;   // compilation error!
  ```
- So far, we have got very basic idea about C++ classes and objects.
Constructor (1)

- In C++ class, a special function, which is automatically called whenever a new object of this class is created, allowing the object to initialize member variables or allocate storage is called **constructor**.
- Constructor function is declared just like a regular member function with the class name, but without any return type (**not even void**).
- Modify the Point class to use constructor, add the following lines in the class declaration:

```cpp
class Point {
    // define a class Point
    private:
        // list of private members ...
    public:
        // define a constructor to initialize members
        Point();
        // list of other member functions
};
```
Constructor (2)

- Definition of the Point class constructor:

  ```cpp
  // define a constructor to initialize members
  // Note that no return type is used
  Point::Point() {
    index=0, tag=0, x=0, y=0; //initialize the private members
  }
  ```

- After defining the constructor, when we define an object variable of Point, its private members are initialized using the constructor.

  ```cpp
  Point p3; // what is index, tag, x, y of p3 at this point?
  ```

- How do we initialize private members using different values at time of definition?

  ```cpp
  // declare another constructor with parameters
  Point(int,char,real,real);
  // define another constructor with parameters
  Point::Point(int idx,char tg,real xc,real yc) {
    index=idx, tag=tg, x=xc, y=yc; //initialize with parameters
  }
  ```
Overloaded constructors

- We have seen the Point class can have two constructors:
  - `Point();`
  - `Point(int, char, real, real);`

- One class can have two functions with the same name, and the objects of Point can be initialized in either of the two ways.
  - `Point p1, p2; // calling the Point() default constructor`
  - `Point p3(0, 'c', 0, 1); // calling the Point(...) constructor`

- The compiler will analyze the types of arguments and matching them to the types of parameters of different function definitions.

- The above example also introduces a special kind constructor: the `default` constructor.
  - The default constructor is the constructor that takes no parameters.
  - The default constructor is called when an object is declared but is not initialized with any arguments.
Destructor

- Destructors are usually used to de-allocate memory and do other cleanup for a class object and its class members when the object is destroyed. Destructor is considered the inverse of constructor function.
- A destructor will have the same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters.
- There is only **one** destructor per class in C++.
- A destructor is called for a class object when that object passes out of scope or is explicitly deleted. An example of destructor definition in Point class:

  ```cpp
  // declare a destructor in class declaration
  ~Point();
  // define the destructor
  Point::~Point() {
    cout << "Destructor called." << endl;
  }
  ```
new and delete in C++

- In C++, the dynamic memory functions are: `new` and `delete`
- The major difference between malloc and free: `new` and `delete` will call the constructor and destructor.
- Using the following constructor and destructor in the Point class:
  ```cpp
  // define another constructor with parameters
  Point::Point() {
    cout << "Constructor called." << endl;
  }
  Point::~Point() { // define the destructor
    cout << "Destructor called." << endl;
  }
  ```
- What will be the output in the main() function call?
  ```cpp
  (point_class_new_delete.cpp)
  void main(void) {
    Point *ptr_p = new Point[2];
    delete[] ptr_p;
    ptr_p =(Point*)malloc(2*sizeof(Point));
    free(ptr_p);
  }
  ```
Overloading functions

- In C++, two different functions can have the same name either:
  - because they have a different number of parameters,
  - because any of their parameters are of a different type.

- See we overload the set_values function for the Point class
  
  ```
  // define the "set_values" method using 4 values
  void Point::set_values(int idx, char tg, real xc, real yc) {
      index=idx, tag=tg, x=xc, y=yc;
  }
  // define the "set_values" method using another object
  void Point::set_values(Point p) {
      index=p.index, tag=p.tag, x=p.x, y=p.y;
  }
  ```

- Overloading is simply defined as the ability of one function to perform different tasks.

- In C++, operators (e.g. +, -, *, /, <<, etc.) can also be overloaded. See training folder for examples (Not covered in this training).
Pointer to class

- Use the same way as pointer to a struct and to access members of a pointer to a class with the member access operator “\-\->”

- As with all pointers, pointers must be initialized before using:

  ```
  Point p1, p2; // calling the Point() constructor
  Point *ptr_p; // define pointer to class
  ptr_p = &p2;  // point ptr_p to p2 object
  p1.set_values(0,'a',0,0); // object p1 use set_values method
  p2.set_values(1,'b',1,1); // object p2 use set_values method
  // access the member function using pointer ptr_p
  ptr_p->set_values(2,'d',1,0);
  ptr_p->print_values();
  p2.print_values();
  ```

- What will be the output? (hint: we used the address of p2)
Inheritance is one of the most important concepts in object-oriented programming.

In C++, a new class can inherit the members of an existing class. The existing class is called the **base class**, and the new class is called the **derived class**.

A derived class can be derived from multiple base classes.

```cpp
// syntax for declaring a derived class
class derived_class: access_specifier base_class_list
{
    AccessSpecifier: public, protected, private
    base-class is the name list of previously defined classes
    If the access-specifier is not used, it is private by default.
}
```

An example of derived class Particle based on Pointe:

```cpp
class Particle: public Point {
};
```

In order for class Particle to access the members in Point: index, tag, x, y, the access specifier needs to be changed to `protected`. 
Access Control and Inheritance

- Base class members in derived class:
  - **public** Inheritance: public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are not accessible.
  - **protected** Inheritance: public and protected members of the base class become protected.
  - **private** Inheritance: public and protected members of the base class become private.
  - Protected or private inheritance is *rarely* used. Most inheritance is **public**.

<table>
<thead>
<tr>
<th>Access</th>
<th>public</th>
<th>protected</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Within Same class</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Derived classes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Outside classes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Implementation of Particle class

In this example, we attempt to create a Particle class based on Point, we will add another attribute: mass of the particle.

```cpp
// declare a derived class Particle based on Point
class Particle: public Point {
    protected:
        real mass;
    public:
        Particle(){ mass=0.0; }
        void set_mass(real m);
        real get_mass();
};
```

// define the set_mass method
```cpp
void Particle::set_mass(real m){
    mass = m;
}
```

// define the get_mass method
```cpp
real Particle::get_mass(){
    return mass;
}
```
Example using the derived class

- Define an object of Particle class and access its methods:

```cpp
int main(void) {
    Particle p; // which constructor is called?
    // calls the base class method (function)
p.set_values(1,'a',0.5,1.0);
p.print_values();
    // calls the derived class method (function)
p.set_mass(1.3);
    // read how to control the format using cout
    cout << "mass of p = " << fixed << setprecision(3)
         << p.get_mass() << endl;
    return 0;
}
```

- The output of the above code on a terminal:

```
$ ./a.out
point a: index = 1, x = 0.5, y = 1
mass of p = 1.300
```
Template and Generic Programming

- Template is a feature of the C++ that allow functions and classes to operate with generic types. There are two kinds of templates: function template and class template.

- Declare a function template:

  // Both expressions have exactly the same meaning behavior.
  template <class identifier> function_declaration;
  template <typename identifier> function_declaration;

- Example of defining a template function:

  // T is a generic "Type"
  template<typename T>
  T add(T a, T b) {
      return a+b;
  }

- C++ provides unique abilities for **Generic Programming** through templates.

- Generic Programming achieved its first major success in C++ with the **Standard Template Library**.
Introduction to STL
(Standard Template Library)

- The Standard Template Library, or STL, is a C++ library of *container classes, algorithms, and iterators*.
- It provides many of the basic algorithms and data structures of computer science.
- STL is a generic library, meaning that its components are heavily parameterized: almost every component in the STL is a template.
- The STL can be categorized into two parts:
  - The Standard Function Library: consists of general-purpose, template based generic functions.
  - The Object Oriented Class Library: a collection of class templates and associated functions.
- STL is now part of the ANSI/ISO C++ Standard.
- We will touch `std::vector` and `std::list` in the training.
  - Fortunately, we can use STL without knowing much about how to write them.
std::vector and std::list

- A std::vector is a collection of objects, all of which have the same type.
- Similar to arrays, vectors use contiguous storage locations for their elements, e.g. elements can also be accessed using offsets on regular pointers to its elements efficiently.
- Unlike arrays, vector can change size dynamically, with their storage being handled automatically by the container.
- Use of std::vector:
  ```cpp
  // include the appropriate header with "using" declaration
  #include<vector>
  using std::vector;
  // define the std::vector objects (variables)
  vector<int> index_vec;  // index_vec holds objects of type int
  vector<double> value_vec; // value_vec holds objects of type double
  vector<Point> point_vec;  // point_vec holds objects of class Point
  ```
An example using std::vector

Below is an example using std::vector to: (1) find a value in an array (2) sort the array

```cpp
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

// using STL to: (1) find a value in an array (2) sort the array
int main() {
    int arr[]={2,3,1,5,4,6,8,7,9,0};
    int *p = find(arr,arr+10,5); // find number "5" using std::find
    p++;
    cout << "The number after 5 is " << *p << endl;
    vector<int> vec (arr,arr+10); // assign the array values to std::vector
    // now sort the array
    sort(vec.begin(),vec.end());
    for(int i=0; i<vec.size(); i++)
        cout << vec[i]<< " ";
    cout << endl;
    return 0;
}
```
Other important C++ concepts

- C++ Polymorphism
  - A call to a member function will cause a different function to be executed depending on the type of object that invokes the function (static polymorphism, dynamic polymorphism)

- C++ Encapsulation
  - Mechanism of exposing only the interfaces and hiding the implementation details from the user. (data hiding)

- C++ Abstraction, etc.

- Refer to C++ text books for further details, e.g.:
  - C++ Primer (Stanley B. Lippman, Josée Lajoie, Barbara E. Moo)
  - Thinking in C++ (Chuck Allison and Bruce Eckel)
  - The C++ Programming Language (Bjarne Stroustrup)
Selected C++ Libraries

- Use existing libraries for your work instead of starting from scratch!
- Generic:
  - Boost
- 3D Graphics:
  - Ogre3D
  - OpenGL
- Math:
  - BLAS and LAPACK
  - UMFPACK
  - Eigen
- Computational geometry
  - CGAL
- Finite Element Method, Finite Volume Method
  - deal.II
  - OpenFOAM, Overture
Exercise 2

- Calculate the result of a constant times a vector plus a vector:
  where $a$ is a constant, $\vec{x}$ and $\vec{y}$ are one dimensional vectors.

\[
\vec{y} \leftarrow a\vec{x} + \vec{y}
\]

1. Complete the code for the vector addition;
2. Change $x$ and $y$ to dynamic arrays
Exercise 3

➢ Complete the C code for matrix multiplication

\[ A \cdot B = C \]

where:

\[ a_{i,j} = i + j \]

\[ b_{i,j} = i \cdot j \]

\[ c_{i,j} = \sum_{k} a_{i,k} \cdot b_{k,j} \]

Also complete the following functions for 2d dynamic array:

- allocate_dynamic_2d_array
- free_dynamic_2d_array