Practical C/C++ programming
Part II

Wei Feinstein
HPC User Services
Louisiana State University

Topics

- Pointers in C
  - Use in functions
  - Use in arrays
  - Use in dynamic memory allocation
- Introduction to C++
  - Changes from C to C++
  - C++ classes and objects
  - Polymorphism
  - Templates
  - Inheritance
- Introduction to Standard Template Library (STL)

What is a pointer?

- A pointer is essentially a **variable** whose value is the address of another variable.
- Pointer “points” to a specific part of the memory.
- Important concept in C programming language. Not recommended in C++, yet understanding of pointer is necessary in Object Oriented Programming
- How to define pointers?

```
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 Operations

(a) Define a pointer variable.

(b) Assign the address of a variable to a pointer.

```
&        /* "address of" operator    */
```

(c) Access the value pointed by the pointer by dereferencing

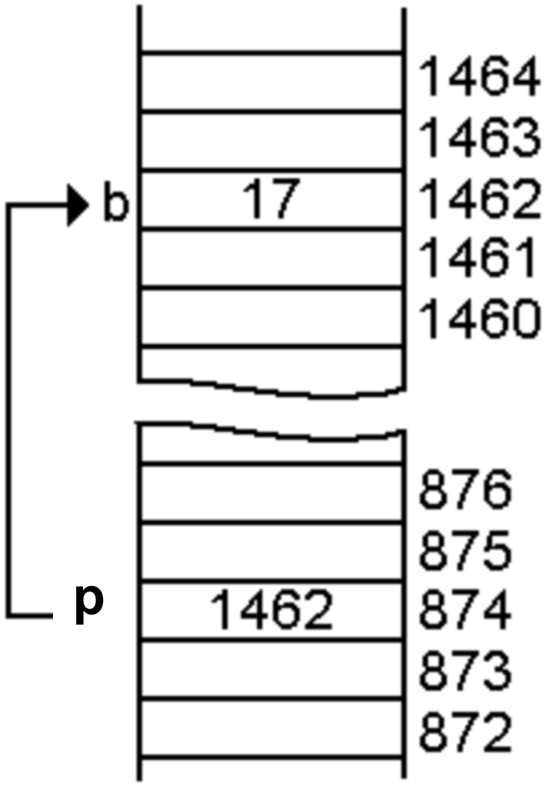
```
*      /* “dereferencing" operator */
```

Examples:

```
int a = 6;
int *ptr;
ptr = &a;    /* pointer p point to a */
*ptr = 10;   /* dereference pointer p reassign a value*/
```

var_name	var_address	var_value
ptr	0x22aac0	0xXXXX
a	0xXXXX	6

Pointer Example




```
int b = 17;
int *p;
/* initialize pointer p */
p = &b;
/*pointed addr and value, its own addr */
printf("pointed addr=%p,p value=%d, p addr=%p", p, *p, &p);
printf("b value=%d, b addr=%p\n",b,&b);
*p = 100;
/* what is b now? */
printf("after *p=100, b=%d", b);
```

```
[wfeinste@mike5 C++]$ gcc pointer.c
[wfeinste@mike5 C++]$ ./a.out
pointed addr=0x7fffdd8901b4   p value p=17, p addr=dd8901a8
b value=17, addr=0x7fffdd8901b4
after *p=100, b=100
```

Never dereference an uninitialized pointer!

- Pointer must have a valid value (address) for dereferencing
- What is the problem for the following code?

```
int *ptr;
*ptr=3;
```

  - undefined behavior at runtime, operate on an unknown memory space.
  - Typically error: "**Segmentation fault**", possible illegal memory operation
- Always initialize your variables before use!

NULL pointer

- A pointer that is assigned NULL is called a null pointer.

```c
/* set the pointer to NULL 0 */
int  *ptr = NULL;
```

- Memory address 0 has special significance when a pointer contains the null (zero) value.
- A good programming practice: before using a pointer, ensure that it is not equal to NULL:

```c
if (ptr != NULL) {
  /* make use of pointer1 */
  /* ... */
}
```

Why Use Pointers

- Pass function arguments by reference
- Efficient, by-reference “copies” of arrays and structures, especially as function parameters
- Array operations (e.g., parsing strings)
- Dynamic memory allocation
- malloc data structures of all kinds, especially trees and linked lists

Function (pass by value)

- In C part I, arguments are **passed by value** to functions: changes of the parameters in functions do **not** change the parameters in the calling functions.
- What are the values of a and b after we called `swap(a, b)`?

```
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 );
}
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 );
}
```

Function (pass by value)

- In C part I, arguments are **passed by value** to functions: changes of the parameters in functions do **not** change the parameters in the calling functions.
- What are the values of a and b after we called `swap(a, b)`?

```
int main() {
    int a = 2;
    int b = 3;
    printf("Before:  a = %d and  b = %d\n", a, b );
    swap( a, b );
    p
}
void
    i
    t
    p                                                    , p2 );
}
```

```
weis-MacBook-Pro:c wei$ gcc pointer_func_swap.c
weis-MacBook-Pro:c wei$ ./a.out
swap by value:
Before:  a = 2 and  b = 3
Swap: a (p1) = 3 and  b(p2) = 2
After:  a = 2 and  b = 3
```

Pointers and Functions (pass by reference)

- Pass by value: parameters called in functions are copies of the callers' passed argument. Caller and called function each has its own copy of parameters
- Solution at this point?
- Use pointers: pass by reference

```
/* 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 or by reference function */
 swap_by_reference( &a, &b );
```

Pointers and Functions (pass by reference)

- Pass by value: parameters called in functions are copies of the caller … as its own …
- Solut…
- Use …

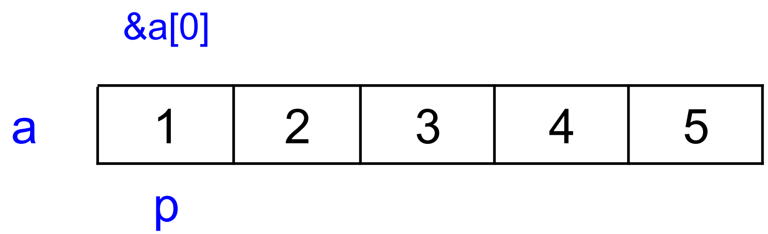
```
weis-MacBook-Pro:c wei$ gcc pointer_func_swap.c
weis-MacBook-Pro:c wei$ ./a.out
swap by reference:
Before:  a = 2 and  b = 3
Swap: a (p1) = 3 and  b(p2) = 2
After:  a = 3 and  b = 2
```

```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 or by reference function */
swap_by_reference( &a, &b );
```

Pointer and 1D Array

- The most frequent use of pointers in C is to efficiently walk arrays.
- Array name is the first element address of the array

```
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] */
```

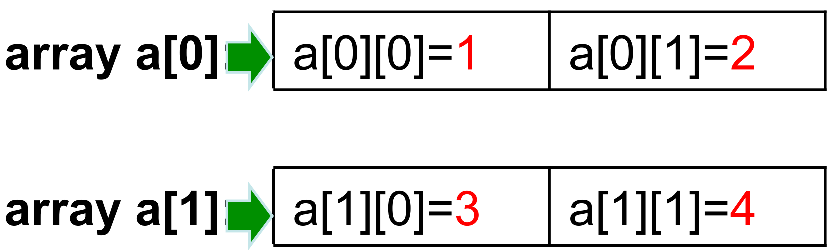


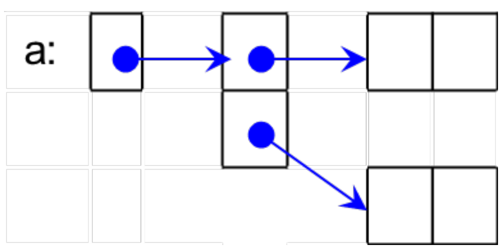
Pointer and 2D Array

- 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]
- a[1] is the address of a[1][0]





Walk through arrays with pointers

```
#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;
    /* pointer to array address */
    i_ptr = a_i;
    f_ptr = a_f;
    /* use ++ 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; }
```

Walk through arrays with pointers

```
#include <stdio.h>
const int MAX = 3;
int main () {
    int  a_i[] = {10, 20, 30};
```

```
weis-MacBook-Pro:C wei$ gcc array_walk.c
weis-MacBook-Pro:C wei$ ./a.out
adr a_i[0] = 0x7fff5254cb1c        adr a_f[0] = 0x7fff5254cb00
val a_i[0] =       10              val a_f[0] =     0.50
adr a_i[1] = 0x7fff5254cb20        adr a_f[1] = 0x7fff5254cb08
val a_i[1] =       20              val a_f[1] =     1.50
adr a_i[2] = 0x7fff5254cb24        adr a_f[2] = 0x7fff5254cb10
val a_i[2] =       30              val a_f[2] =     2.50
```

```
    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

- When the size of an array is unknown at compiling time, pointers are used 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:

```
/* 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 dynamic 1D array

```c
/* 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;
}
```

Example of dynamic 1D 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_arr
    /* ou
    for (
        i                                                        rray */
        p
    }
    printf("\n");
    free((void*)i_array); /* free memory after use*/
    return 0;
}
```

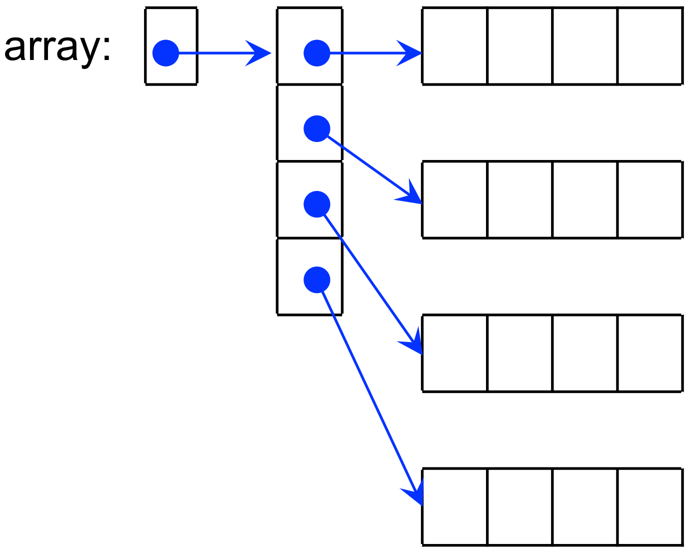
```
weis-MacBook-Pro:C wei$ gcc dynamic_1d_array.c
weis-MacBook-Pro:C wei$ ./a.out
Input the number of elements in the array:
10
0 1 2 3 4 5 6 7 8 9
```

Dynamic 2D array

Use dynamic 2D array (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)

Topics

- Pointers in C
  - Use in functions
  - Use in arrays
  - Use in dynamic memory allocation
- Introduction to C++
  - Changes from C to C++
  - C++ classes and objects
  - Polymorphism
  - Templates
  - Inheritance
- Introduction to Standard Template Library (STL)

C++ Programming Language

- A general-purpose object-oriented programming language. An extension to C language.
- Collection of predefined classes, such as STL, a C++ library of container classes, algorithms, and iterators.
- Polymorphism: Operations or objects behave differently in different contexts
  - static polymorphism: overloading
  - dynamic polymorphism: inheritance, overriding
- Encapsulation: Expose only the interfaces and hide the implementation details
- Abstraction: Provide a generalization by displaying only essential info and hiding details

From C to C++

- Major difference: C is function-driven while C++ is **object oriented**.
- Some minor C++ features over C
  - "//" for comments
  - **`using namespace std:`** Use standard C++ libraries
  - **`cout`** `<<` (insertion operator): output to the screen
  - **`cin`** `>>` (extraction operator): input from the keyboard
  - Variables can be declared anywhere inside the code
  - Can use reference for a variable (preferable) instead of pointer
  - Memory manipulation: **`new`** and **`delete`**
- To compile a C++ program:

```
$ g++ sample.cpp
$ icpc sample.cpp
```

C++ Standard Library and std namespace

- Standard library names are defined using the following C++ syntax:

```
#include <cstdlib> // instead of stdlib.h
```

- All C++ standard library names, including the C library names are defined in the namespace `std`.
- Use one of the following methods:
  - Specify the standard namespace, for example:
    ```
    std::printf("example\n");
    ```
  - C++ keyword using to import a name to the global namespace:
    ```
    using namespace std;
    printf("example\n");
    ```
  - Use the compiler option `--using_std`.

Example

```
#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++) {  // Local declaration of 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: an ***alias*** for the variable, 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:

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

- Example using C++ reference as function parameters (see ***ref.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;
}
void swap (int& a, int& b) {
    int t;
    t=a,a=b,b=t;
}
```

```
weis-MacBook-Pro:c++ wei$ c++ ref.cpp
weis-MacBook-Pro:c++ wei$ ./a.out
before: x=3 y=4
after : x=4 y=3
```

C++ class definition and modifiers

- class_name is a valid identifier for the class
- The body of the declaration contains members, which can either be data or function declarations, and access modifiers.
- Access specifiers/modifiers are keywords to set accessibility of classes, methods and other members:
  - `private: // accessible only from within class or their "friends"`
  - `public:  // accessible from outside and within the class through an object of the class`
  - `protected: // accessible from outside the class ONLY for derived classes.`
- By default, all members of a class is `private` unless access specifier is used.

Class example: Point class

```
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:
    void set_values(int,char,real,real);
    void print_values();
};
void Point::set_values(int idx,char tg,real xc,real yc) {
    index=idx, tag=tg, x=xc, y=yc;
}
void Point::print_values() {
   cout << "point " << tag << ": index = " << index
        << ", x = " << x << ", y = " << y << endl;
}
```

Point Class

- `private` members of Point: **`index, tag, x, y`** cannot be accessed from outside the Point 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:

```
// 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 :-)
}
```

Class Objects

- An object is an instance of a class.
- To declare objects of a class is similar to 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
```

- 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
```

Constructor (1)

- Constructor is automatically called whenever a new object of a class is created, to initialize member variables or allocate storage.
- 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:

```
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:

```
// 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.

```
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?

```
// 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

- 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.
- 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 analyze the types of arguments and match them to the types of parameters of different function definitions.

Destructor

- Destructors are usually used to de-allocate memory, cleanup a class object and its class members when the object is destroyed.
- 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 when that object passes out of scope or is explicitly deleted. An example of destructor definition in Point class:

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

new and delete in C++

- Used for dynamic memory allocation
- **new** and **delete** call the constructor and destructor, compared to malloc and free (C)
- Using the following constructor and destructor in the Point class:

```
// 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? (***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);
}
```

new and delete in C++

- Used for dynamic memory allocation
- **`new`** and **`delete`** call the constructor and destructor, compared to malloc and free (C)
- Using the … class:

```
// define a
Point::Poi
    cout <<
}
Point::~Poi
    cout <<
}
```

```
weis-MacBook-Pro:c++ wei$ c++ point_class_new_delete.cpp
weis-MacBook-Pro:c++ wei$ ./a.out
Constructor called.
Constructor called.
Destructor called.
Destructor called.
using malloc:
```

- What will be the output in the main() function call? (***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

- Functions with a same name but different in:
  - Number of parameters
  - Type of parameters
- One function to perform different tasks.
- Overload set_values function of 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;
}
```

- Operators (e.g. **+, -, *, /, <<, etc.**) can also be overloaded. See training folder for examples (Not covered in this training).

Operator Overloading

```
…
class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)  {real = r;   imag = i;}
    Complex operator + (Complex const &obj) {
         Complex res;
         res.real = real + obj.real;
         res.imag = imag + obj.imag;
         return res;}
    void print() { cout << real << " + i" << imag << endl; }
};
int main(){
    Complex c1(10, 5), c2(
    Complex c3 = c1 + c2;
    c1.print(); c2.print()
    cout<<"after calling c
    c3.print();
}
```

```
[wfeinste@mike5 c++]$ c++ overload_operator.cpp
[wfeinste@mike5 c++]$ ./a.out
10 + i5
2 + i4
after calling complex + overload
12 + i9
```

Template for Generic Programming

- C++ feature allows functions and classes to operate with generic types. C++ provides unique abilities for Generic Programming through templates.
- Two types of templates:
  - function template
  - class template

```
template <typename identifier> function_declaration;
template <class    identifier> class class_declaration;
```

- Example of defining a template function:

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

Function Template

```
…
template <typename T>
inline T const Max (T const a, T const b) {
   return a < b ? b:a;
}
int main () {
   int i = 39;
   int j = 20;
   cout << "Max("<<i<<","<<j<<"): " << Max(i,j) << endl;

   double f1 = 13.5;
   double f2 = 20.7;
   cout << "Max("<< f1 <<","<<f2<<"): " << Max(f1,f2) << endl;

   string s1 = "Hello";
   string s2 = "World";
   cout << "Max("<< s1 <<'
   return 0;
}
```

```
[wfeinste@mike5 c++]$ c++ template_max.cpp
[wfeinste@mike5 c++]$ ./a.out
Max(39,20):  39
Max(13.5,20.7): 20.7
Max(Hello,World): World
```

Class Template

```
…
using namespace std;

template <class T>
class Stack {
   private:
      vector<T> elems;
   public:
      void push(T const&)
      void pop();
      T top() const;
      bool empty() const
         return elems.empty();
      }
};
template <class T>
void Stack<T>::push (T const& elem) {
   // append copy of passed element
   elems.push_back(elem);
}

template <class T>
void Stack<T>::pop () {
   if (elems.empty()) {
      throw out_of_range("Stack<>::pop():
empty stack");
   }
   // remove last element
   elems.pop_back();
}
```

```
template <class T>
T Stack<T>::top () const {
   if (elems.empty()) {
      throw out_of_range("Stack<>::top(): empty
stack");
   }

   // return copy of last element
   return elems.back();
}

int main() {
   try {
      Stack<int> intStack
      Stack<string> stringStack;
      intStack.push(7);
      cout << intStack.top() <<endl;

      // manipulate string stack
      stringStack.push("hello");
      cout << stringStack.top() << std::endl;
      stringStack.pop();
      stringStack.pop();
   } catch (exception const& ex) {
      cerr << "Exception: " << ex.what()
<<endl;
      return -1;
   }
}
```

Class Template

```
…
using namespace std;

template <class T>
class Stack {
   private:
      vector<T>
   public:
      void push
      void pop(
      T top() c
      bool empty() const
         return elems.empty();
      }
};
template <class T>
void Stack<T>::push (T const& elem) {
   // append copy of passed element
   elems.push_back(elem);
}

template <class T>
void Stack<T>::pop () {
   if (elems.empty()) {
      throw out_of_range("Stack<>::pop():
empty stack");
   }
   // remove last element
   elems.pop_back();
}
```

```
template <class T>
T Stack<T>::top () const {
   if (elems.empty()) {
                    ck<>::top(): empty

                    ment

int main() {
   try {
      Stack<int> intStack
      Stack<string> stringStack;
      intStack.push(7);
      cout << intStack.top() <<endl;

      // manipulate string stack
      stringStack.push("hello");
      cout << stringStack.top() << std::endl;
      stringStack.pop();
      stringStack.pop();
   } catch (exception const& ex) {
      cerr << "Exception: " << ex.what()
<<endl;
      return -1;
   }
}
```

```
weis-MacBook-Pro:c++ wei$ c++ template_class.cpp
weis-MacBook-Pro:c++ wei$ ./a.out
7
hello
Exception: Stack<>::pop(): empty stack
```

Derived Class - Inheritance

- In C++ a new class can inherit the members of an existing class.
  - base class: the existing class
  - derived class: new class
- A derived class can be derived from multiple base classes.

```
// syntax for declaring a derived class
class derived_class: access_specifier base_class_list
```

Access_specifier: `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:

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

Access Control and Inheritance

Member accessibility depends on modifiers used in base class:

Access	**public**	**protected**	**private**
Base class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

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

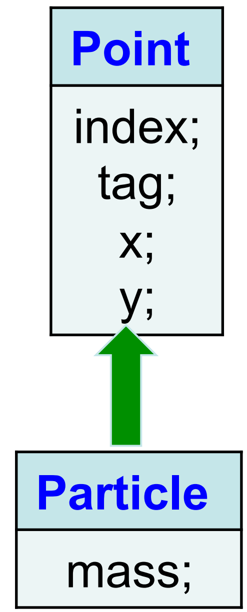
Class example: Point class

```
class Point {  //define a class Point
    // private:
    protected:
        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:
        Point() {index=0, tag=0, x=0, y=0; }
        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;
    }
    void Point::print_values() {
       cout << "point " << tag << ": index = " << index
            << ", x = " << x << ", y = " << y << endl;
    }
}
```

Implementation of Particle class

- Create a Particle class based on Point
- Add another attribute: mass of the particle.

```
// declare a derived class Particle based on Point
class Particle: public Point {
    protected:
        real mass;
    public:
        Particle(){index=0, tag=0, x=0, y=0; mass=0.0; };
        void set_mass(real);
        real get_mass();
};
// define the set_mass method
void Particle::set_mass(real m){
    mass = m;
}
// define the get_mass method
real Particle::get_mass(){
    return mass;
}
```



Example using the derived class

- Use Particle class and access its methods:

```
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;
}
```

```
weis-MacBook-Pro:c++ wei$ c++ point_class_derived.cpp
weis-MacBook-Pro:c++ wei$ ./a.out
point  : index = 0, x = 0, y = 0
mass of p = 1.300
```

Topics

- Pointers in C
  - Use in functions
  - Use in arrays
  - Use in dynamic memory allocation
- Introduction to C++
  - Changes from C to C++
  - C++ classes and objects
  - Polymorphism
  - Templates
  - Inheritance
- Introduction to Standard Template Library (STL)

Introduction to STL (Standard Template Library)

- The Standard Template Library, or STL, is a C++ library of container classes, algorithms, and iterators.
- 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 a generic library, meaning that its components are heavily parameterized: almost every component in the STL is a template to achieve generic programming.
- STL is now part of the ANSI/ISO C++ Standard.

std::vector

- 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:

```
// 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

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

```
#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> my_vec (arr,arr+10); // assign the array values to
std::vector
    // now sort the array
    sort(my_vec.begin(), my_vec.end());
    for(int i=0; i<vec.size(); i++)
        cout << m_vec[i]<< " ";
    cout << endl;
    return 0;
}
```

An example using std::vector

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

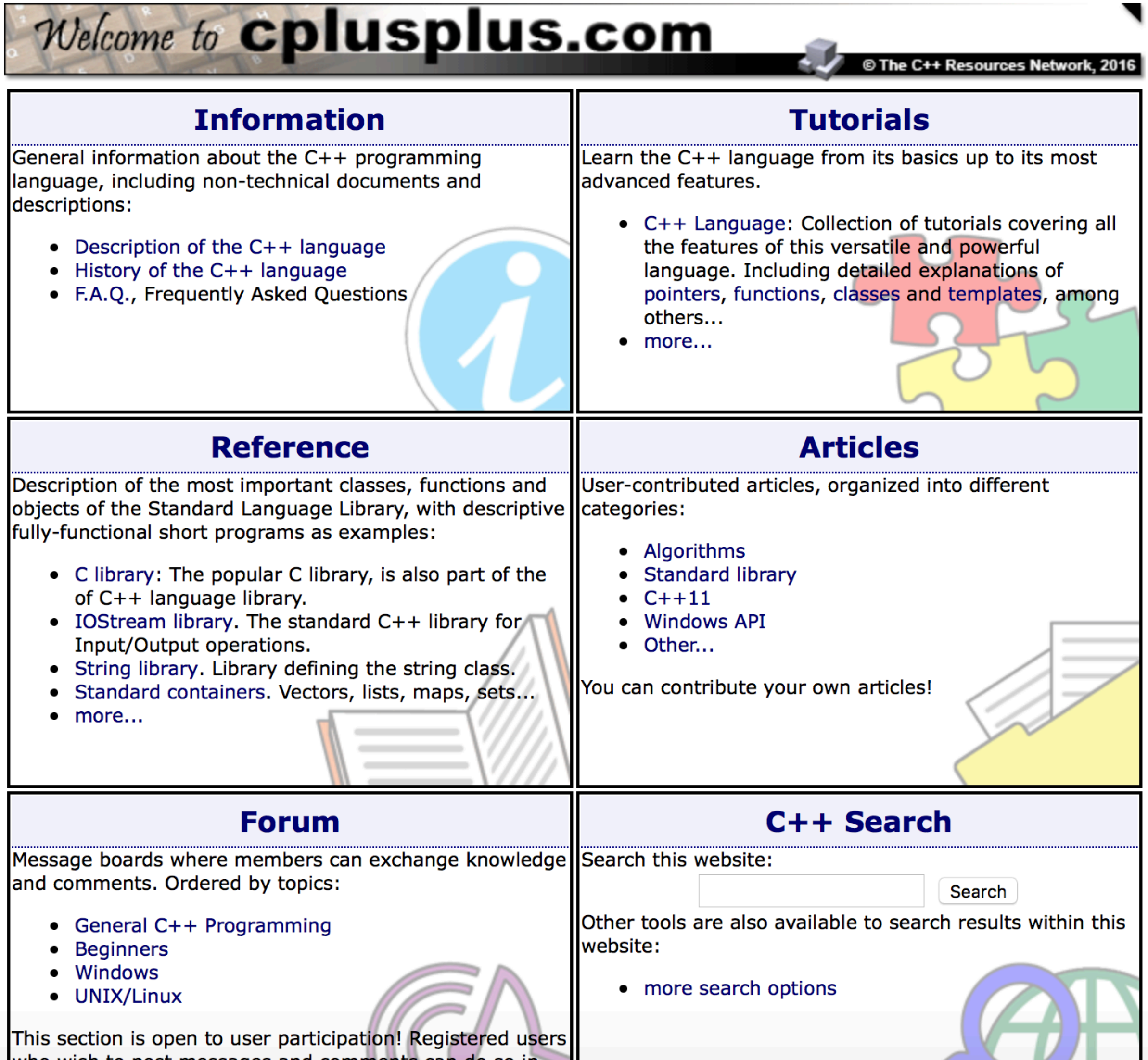
```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
// using STL to: (1) find a value in an array (2) sort the array
int main() {
    int arr[]={2,3,1,5,4,6,8,7,9,0};
    int *p = find(arr,arr+10,5); // find number "5" using std::find
    p++;
    cout << "The number after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

```
weis-MacBook-Pro:c++ wei$ c++ stl_vector.cpp
weis-MacBook-Pro:c++ wei$ ./a.out
The number after 5 is 4
Now the length of the vector is:12
-1 0 1 2 3 4 5 6 7 8 9 11
use vector iterator
-1 0 1 2 3 4 5 6 7 8 9 11
```

Selected C++ Libraries

- Use existing libraries instead of reinvent the wheel
- 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

C++ Resources



Welcome to cplusplus.com

© The C++ Resources Network, 2016

Information

General information about the C++ programming language, including non-technical documents and descriptions:

- Description of the C++ language
- History of the C++ language
- F.A.Q., Frequently Asked Questions

Tutorials

Learn the C++ language from its basics up to its most advanced features.

- C++ Language: Collection of tutorials covering all the features of this versatile and powerful language. Including detailed explanations of pointers, functions, classes and templates, among others...
- more...

Reference

Description of the most important classes, functions and objects of the Standard Language Library, with descriptive fully-functional short programs as examples:

- C library: The popular C library, is also part of the of C++ language library.
- IOStream library. The standard C++ library for Input/Output operations.
- String library. Library defining the string class.
- Standard containers. Vectors, lists, maps, sets...
- more...

Articles

User-contributed articles, organized into different categories:

- Algorithms
- Standard library
- C++11
- Windows API
- Other...

You can contribute your own articles!

Forum

Message boards where members can exchange knowledge and comments. Ordered by topics:

- General C++ Programming
- Beginners
- Windows
- UNIX/Linux

This section is open to user participation! Registered users who wish to post messages and comments can do so in

C++ Search

Search this website:

Search

Other tools are also available to search results within this website:

- more search options

http://www.cplusplus.com