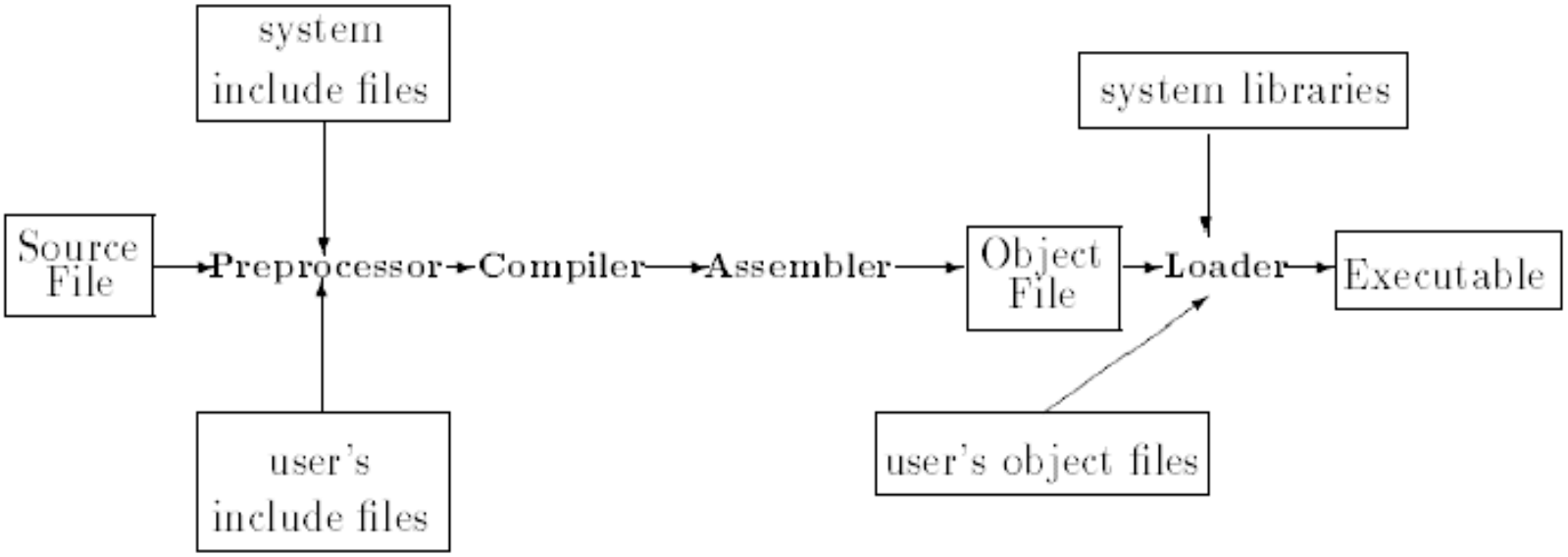
Short Notes on C/C++

- Structure of a program
  - See ~zxu2/Public/ACMS40212/C++_basics/basics.cpp



Compilation Stages

  - To see how the code looks after pre-processing, type icc –A –E basics.cpp

Program Structure

- **Preprocessor directive**
  - It performs instructions compiler performed before the program is compile.
  - The program “basics.cpp” have following directives.

```
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
```

- Every C/C++ program contains exactly one block of code called *main* function.
- **Declarations**
  - Declarations define identifiers and must precede any **statements** that references these identifiers.
  - The program “basics.cpp” have following declaration

```
int answer, i =4, j = 5, k(4);
```

- **Statements**
  - Specify the operations to be performed.
  - The program “basics.cpp” have following statements

```
std::cout << "Hello World!";
answer = mean(i,j);
printf("The mean of %d and %d is %d\n", i, j, answer);
```

Variables and Literals

- The available scalar types are *char, short, int, long, float, double* and *long double* etc.

```
unsigned int i, j;
float f = 3.14;
double  pi(3.14);  // we can also initialize value of variable
                   //in C++ using this way.
i = (unsigned int)f;
```

Aggregates

1. Variables of the same type can be put into arrays or multi-D arrays, e.g.,
   char letters[50], values[50][30][60];
   Remark: C has no subscript checking; if you go to the end of an array, C won't warn you.
2. Variables of different types can be grouped into a *structure*.

```
typedef struct {
        int age;
        int height;
        char surname[30];
} person;
…
person fred;
fred.age = 20;  //”.” operator is to access the member variable.
```

Remark: variables of structure type can not be compared.

Do not do:

```
person fred, jane;
…
if(fred == jane)
{
     printf(“the outcome is undefined”);
}
```

Constructions

1. Selection

```
...
if (i==3) /* checking for equality; `!=' tests for inequality */
{
    j=4;
}
else{
   j=5;
   k=6;
}
...
```

```
...
/*  Switch is used for multiple-selection decision making.
The values that the switching variable is compared with case labels, which
  have to be constants, or `default'.
 */

switch(i){
case 1: printf("i is one\n");
      break; /* if break wasn't here, this case will
                fall through into the next.
              */
case 2: printf("i is two\n");
      break;
default: printf("i is neither  one nor two\n");
      break;
}
...
```

Loops

- while loop

```
…
while(i<30){    /* test at top of loop */
  something();
…
}
```

- do/while loop

```
…
do {
   something();
} while (i<30); /* test at bottom of loop */
…
```

- for loop

```
…
for(i=0; i<5; i=i+1){
   something();
}
…
```

Preprocessor directives

- Preprocessor directives are lines included in the code of programs preceded by a hash sign (#) and are processed by preprocessor of compiler.
- No semicolon (;) is expected at the end of a preprocessor directive.

1. macro definitions (#define, #undef)
   - syntax: #define identifier replacement
   - When the preprocessor encounters this identifier, it replaces any occurrence of identifier in the rest of the code by replacement.
   - **Example**.

```
#define TABLE_SIZE 100
#define getmax(a,b) ((a)>(b)?(a):(b))
#define sqr(x)          ((x)*(x))
main()
{
    int table1[TABLE_SIZE], x = 10, y;
    y= getmax(x,2);
    y= sqr(11 - 5);
}
```

2. Conditional inclusions (#ifdef, #ifndef, #if, #endif, #else and #elif)

- These directives allow to include or discard part of the code of a program if a certain condition is met.

```
#define TABLE_SIZE 100

main(){
   #ifdef TABLE_SIZE
   int table[TABLE_SIZE];
   #endif
}
```

```
#ifndef TABLE_SIZE
#define TABLE_SIZE 100
#endif

main(){
   int table[TABLE_SIZE];
}
```

```
#if TABLE_SIZE>200
  #undef TABLE_SIZE
  #define TABLE_SIZE 200

#elif TABLE_SIZE<50
  #undef TABLE_SIZE
  #define TABLE_SIZE 50

#else
  #undef TABLE_SIZE
  #define TABLE_SIZE 100
#endif

main(){
   int table[TABLE_SIZE];
}
```

See also:
ACMS40212/C_basics/struct_sample

3. Source file inclusion (#include)
   - When the preprocessor finds an #include directive it replaces it by the entire content of the specified header or file.
   - #include <header> // This is used to include
     //headers provided by the compiler.
   - #include "file" // The file is searched for in an
     // implementation-defined manner,
     //which generally includes the current path.
4. Predefined macro names
   - __LINE__ // Integer value representing
     //the current line in the source code file being compiled.
   - __cplusplus // An integer value. All C++ compilers
     //have this constant defined
   - …

See also: http://www.cplusplus.com/doc/tutorial/preprocessor/

Functions

- A *function* is a group of statements that together perform a task.
- A functions generally require a prototype which gives basic structural information: it tells the compiler what the function will return, what the function will be called, as well as what arguments the function can be passed.

```
#include <stdlib.h>   /* Include rand() */
int main()
{
   int a = rand(); /* rand is a standard function that all compilers have */
}
```

- The form of a function definition is:

```
return_type function_name (formal argument list )
{
Declarations;  // good practice to have declarations at beginning
Statements;
}
```

```
double mean(double x, double y)
{
    double tmp;

     tmp = (x + y)*0.5;
     return tmp;
}
```

C Input/Output

- Input: feed some data into a program. An input can be given in the form of a file or from the command line.
- Output: display some data on screen, printer, or in any file.
- C programming provides a set of built-in functions for input/output.
- C programming treats all the devices as files.
- The following three files are automatically opened when a program executes to provide access to the keyboard and screen.

Standard File	File Pointer	Device
Standard input	stdin	Keyboard
Standard output	stdout	Screen
Standard error	stderr	Your screen

scanf() and printf() of C Language

- The int **scanf(const char *format, ...)** function reads the input from the standard input stream stdin and scans that input according to the format provided.
- The int **printf(const char *format, ...)** function writes the output to the standard output stream stdout and produces the output according to the format provided.
- The **format** can be a simple constant string, but %s, %d, %c, %f, etc. can be specified to print or read strings, integer, character or float respectively.

There are many other I/O functions.

- int fscanf(FILE *stream, const char* format, ...);
  // Reads the data from file stream stream
- int fprintf(FILE* stream, const char* format, ...);
  // Writes the results to a file stream

```
/* scanf_printf.c */

#include <stdio.h>
int main( ) {

  char str[100];
  int i;

  printf( "Enter a value :");
  scanf("%s  %d", str, &i);

  printf( "\nYou entered: %s %d ", str, i);

  return 0;
}

/* see also example  in C_basics/struct_sample */
```

C++ IO Streams

- C++ supports all input/output mechanisms that C includes.
- A 'stream' is internally nothing but a series of characters. The characters may be either normal characters (char) or wide characters (wchar_t). Streams provide users with a universal character-based interface to any type of storage medium (for example, a file), without requiring to know the details of how to write to the storage medium.
- A program can either insert or extract characters to/from stream.
- Streams work with built-in data types, and users can make user-defined types work with streams by overloading the insertion operator (<<) to put objects into streams, and the extraction operator (>>) to read objects from streams.
- The stream library's unified approach makes it very friendly to use. Using a consistent interface for outputting to the screen and sending files over a network makes life easier.

C++ Standard Stream Objects for Console I/O

- These are declared in the <iostream> header
- The class *ostream* is defined with operator << ("put to") to handle output of the build-in types. Object of *ostream* corresponds to the C stream stdout.
  - extern std::ostream cout; // is defined in <iostream>
  - The global objects *std::cout* control output to a stream buffer of implementation-defined type.
  - *std::cout* is guaranteed to be initialized.
- The class *istream* is defined with operator >> ("get from") to handle input of the build-in types. Object of istream corresponds to the C stream stdin.
  - extern std::ostream cin; // is defined in <iostream>
  - The global objects *std::cin* control input to from stream buffer of implementation-defined type.
  - *std::cin* is guaranteed to be initialized.

```
// i/o example console_io.cpp

#include <iostream>
using namespace std;

int main ()
{
  int i;
  cout << "Please enter an integer value: ";
  cin >> i;
  cout << "The value you entered is " << i;
  cout << " and its double is " << i*2 << ".\n";
  return 0;
}

// see also: http://www.cplusplus.com/forum/articles/6046/
```

C++ Input/output with Files

- Following classes are used to perform output and input of characters to/from files:
  - *ofstream*: Stream class to write on files
  - *ifstream*: Stream class to read from files
  - fstream: Stream class to both read and write from/to files.
- Open a file
  - In order to open a file with a stream object we use its member function open:

  open (filename, mode);
  - Example

  ofstream myfile;

  myfile.open ("example.bin", ios::out | ios::app | ios::binary);

  // ios is the base class for all stream classes.
- Closing a file
  - myfile.close();

// see examples: f_iostream.cpp, binary_rw.cpp

Pointers

- A variable is a memory location which can be accessed by the identifier (the name of the variable).
  - int *k*; /* the compiler sets aside 4 bytes of memory (on a PC) to hold the value of the integer. It also sets up a symbol table. In that table it adds the symbol *k* and the relative address in memory where those 4 bytes were set aside. */
  - *k* = 8; /*at run time when this statement is executed, the value 8 will be placed in that memory location reserved for the storage of the value of k. */
- With *k*, there are two associated values. One is the value of the integer, 8, stored. The other is the "value" or address of the memory location.
- The variable for holding an address is a pointer variable.

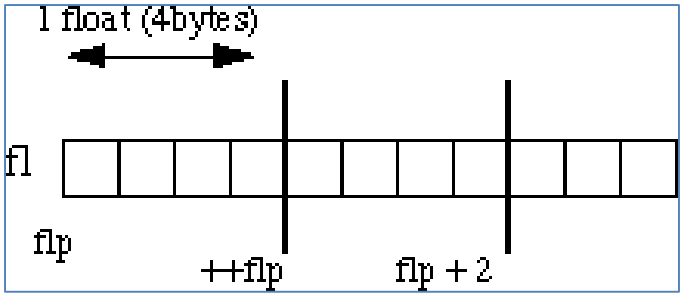
  int *ptr; /*we also give pointer a type which refers to the type of data stored at the address that we will store in the pointer. "*" means pointer to */

```
ptr = &k;   /* & operator retrieves the address of k */
*ptr = 7;    /* dereferencing operator "*" copies 7 to the address pointed to by
                ptr */
```

See code: ACMS40212/C_basics/pointer_basics.c

- Pointers and 1D arrays

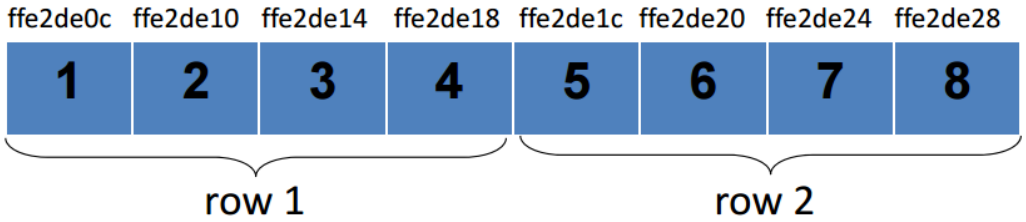
```
float  a[100],  *flp;
flp = &(a[0]);   /* or flp = a; */ // Point flp to the first element in a[]
/* now increment flp to point to successive elements */
for(int i =0; i < 100; i++)
{

     printf("*flp is %f\n", * flp);
    flp++;   /*or flp += 1; */  // flp is incremented by the length of an float
                                  // and points to the next float, a[1], a[2] etc.
}
```



See code: ACMS40212/C_basics/pointer_array.c

- Multidimensional arrays and pointers

1. Array elements are stored row by row. Hence a 2D array (i.e. int myMatrix[2][4] ={{1,2,3,4}, {5,6,7,8}};) is really a 1D array, each of whose element is itself an array.



2. myMatrix: pointer to the first element of the 2D array;

   // myMatrix can be thought as a pointer to a pointer to int.

   myMatrix[0]: pointer to the first row of the 2D array;

   myMatrix[1]: pointer to the second row of the 2D array.

3. myMatrix[i][j] is same as: *(myMatrix[i] + j),

   *(&myMatrix[0][0] + 4*i + j), (*(myMatrix + i))[j], and …

4. int *b[30]; //declares an array of 30 pointers to ints.

Pointers and Text Strings

- Text strings in C have been implemented as arrays of characters, with the last byte in the string being a zero, or the null character '\0'.

```
static const char *myFormat = "Total Amount Due: %d";
//The variable myFormat can be viewed as an array of 21 characters.
```

- An initialized array of strings would typically be done as follows:

```
static const char *myColors[] = {
  "Red", "Orange", "Yellow", "Green", "Blue", "Violet" };
```

- Using a pointer avoids copies of big structures.

```
typedef struct {
        int age;
        int height;
        char surname[30];
} person;
int sum_of_ages(person *person1, person *person2)
{
    int sum; // a variable local to this function
    /* Dereference the pointers, then use the `.' operator to get the fields */
     sum = (*person1).age + (*person2).age;
     /* or    use the notation "->":
         sum = person1->age + person2->age;   */
     return sum;
}

int main()
{
      person fred, jane;
      int sum;
       …
       sum = sum_of_ages(&fred, &jane);
}
```

See also: ACMS40212/C_basics/struct_sample

Dynamic Memory Allocation in C/C++

Motivation

```
 /*   a[100]  vs. *b or *c     */
Func(int array_size)
{
    double  k, a[100], *b, *c;
    b = (double *) malloc(array_size * sizeof(double));  /* allocation in C*/
    c = new double[array_size];  /* allocation in C++ */
    …
}
```

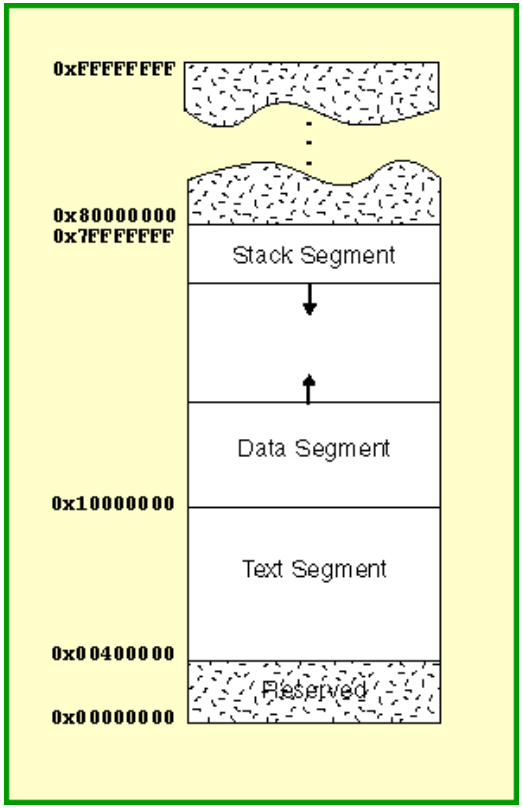
- The size of the problem often can not be determined at “compile time”.
- Dynamic memory allocation is to allocate memory at “run time”.
- Dynamically allocated memory must be referred to by pointers.

Remark: use debug option to compile code ~zxu2/Public/dyn_mem_alloc.cpp and use debugger to step through the code.

Also ~zxu2/Public/dyn_mem_alloc_CPP_ver.cpp

icc –g dyn_mem_alloc.cpp

Stack vs Heap



When a program is loaded into memory:

- Machine code is loaded into **text** segment
- **Data** segment: global, static and constants data are stored in the data segment
- **Stack** segment allocates memory for automatic variables within functions, and is used for passing arguments to the function etc.
- **Heap** segment is for dynamic memory allocation
- The size of the text and data segments are known as soon as compilation is completed. The stack and heap segments grow and shrink during program execution.

Memory Allocation/Free Functions in C

- void *malloc(size_t number_of_bytes)
  -- allocate a contiguous portion of memory
  -- it returns a pointer of type void * that is the beginning place in memory of allocated portion of size number_of_bytes.
- void free(void * ptr);
  -- A block of memory previously allocated using a call to malloc, calloc or realloc is deallocated, making it available again for further allocations.

“void” type: was introduced to make C syntactically consistent. The main reason for void is to declare functions that have no return value. The word "void" is therefore used in the sense of "empty" rather than that of "invalid".

A variable that is itself declared void is useless. Void pointers (type void *) are a different case, however. A void pointer is a generic pointer; any pointer can be cast to a void pointer and back without any loss of information. Any type of pointer can be assigned to (or compared with) a void pointer, without casting the pointer explicitly.

Memory Allocation/Free Functions in C++

C++:

- “new” operator
  -- pointer = new type;
  -- pointer = new type [number_of_elements];
  It returns a pointer to the beginning of the new block of memory allocated.
- “delete” operator
  -- delete pointer;
  -- delete [] pointer;

Example 1

```
double *Func()
/* C++ version */
{
        double *ptr;
        ptr = new double;
        *ptr = -2.5;
        return ptr;
}
```

```
double *Func_C()
/* C version */
{
        double *ptr;
        ptr = (double *)malloc(sizeof(double));
        *ptr = -2.5;
        return ptr;
}
```

Name	Type	Contents	Address
ptr	double pointer	0x3D3B38	0x22FB66

Memory heap (free storage we can use)	
…	
0x3D3B38	-2.5
0x3D3B39	

Example 2

```
Func() /* C++ version , see also  zxu2/Public/dyn_array.c */
{
        double *ptr, a[100];
        ptr = new double[10];   /* in C, use: ptr = (double *)malloc(sizeof(double)*10); */
        for(int i = 0; i < 10; i++)
            ptr[i] = -1.0*i;
        a[0] = *ptr;
        a[1] = *(ptr+1);   a[2] = *(ptr+2);
}
```

- **Illustration**

Name	Type	Contents	Address
ptr	double array pointer	0x3D3B38	0x22FB66

Memory heap (free storage we can use)	
…	
0x3D3B38	0.0
0x3D3B39	-1.0
…	

Example 3

- Static array of dynamically allocated vectors

```
Func() /* allocate a contiguous memory which we can use for 20 ×30 matrix */
{
      double *matrix[20];
      int   i, j;
      for(i = 0; i < 20; i++)
         matrix[i] = (double *) malloc(sizeof(double)*30);

      for(i = 0; i < 20; i++)
      {
           for(j = 0; j < 30; j++)
                matrix[i][j] =  (double)rand()/RAND_MAX;
      }

}
```

Example 4

- Dynamic array of dynamically allocated vectors

```
Func() /* allocate a contiguous memory which we can use for 20 ×30 matrix */
{ // see C++ example Public/dyn_2dmatrix.cpp
      double **matrix; //double** is a pointer to an double*.
      int   I, j;

      matrix = (double **) malloc(20*sizeof(double*));
      for(I = 0; I < 20; I++)
         matrix[I] = (double *) malloc(sizeof(double)*30);

      for(I = 0; I < 20; I++)
      {
           for(j = 0; j < 30; j++)
                matrix[I][j] =  (double)rand()/RAND_MAX;
      }
}
```

Example 5

- Another way to allocate dynamic array of dynamically allocated vectors

```
Func() /* allocate a contiguous memory which we can use for 20 ×30 matrix */
{ // Can you write a C++ version?
      double **matrix;
      int   i, j;

      matrix = (double **) malloc(20*sizeof(double*));
      matrix[0] = (double*)malloc(20*30*sizeof(double));

      for(i = 1; i < 20; i++)
         matrix[i] = matrix[i-1]+30;

      for(i = 0; i < 20; i++)
      {
           for(j = 0; j < 30; j++)
                matrix[i][j] =  (double)rand()/RAND_MAX;
      }
}
```

Release Dynamic Memory

```
Func()
{
    int *ptr, *p;
    ptr = new int[100];
    p = new int;
    delete[] ptr;
    delete  p;
}
```

References

- Like a pointer, a *reference* is an alias for an object (or variable), is usually implemented to hold a machine address of an object (or variable), and does not impose performance overhead compared to pointers.
  - The notation **X&** means “reference to **X**”.
- Differences between reference and pointer.
  1. A reference can be accessed with exactly the same syntax as the name of an object.
  2. A reference always refers to the object to which it was initialized.
  3. There is no “null reference”, and we may assume that a reference refers to an object.

```
void f()   // check the code ~zxu2/Public/reference.cpp
{
    int var = 1;
    int& r{var}; // r and var now refer to the same int. same as: int& r = var;
    int x = r;      // x becomes 1
    r = 2;          // var becomes 2
     ++r;           // var becomes 3
     int  *pp = &r;         // pp points to var.
}

void f1()
{
    int var = 1;
    int& r{var}; // r and var now refer to the same int
    int& r2;      // error: initialization missing
}
```

Remark:

1. We can not have a pointer to a reference.
2. We can not define an array of references.

Functions and passing arguments

1. **Pass by value** //see ~zxu2/Public/Func_arguments

```
1.  #include<iostream>
2.  void foo(int);

3.  using namespace std;
4.  void foo(int y)
5.  {
6.       y = y+1;
7.       cout << "y + 1 = " << y << endl;
8.  }
9.
10. int main()
11. {
12.     foo(5); // first call
13.
14.     int x = 6;
15.     foo(x); // second call
16.     foo(x+1); // third call
17.
18.     return 0;
19. }
```

When foo() is called, variable y is created, and the value of 5, 6 or 7 is copied into y. Variable y is then destroyed when foo() ends.

Remark: Use debug option to compile the code and use debugger to step through the code.
icc -g pass_by_val.cpp

2. Pass by address (or pointer)

```
1.  #include<iostream>
2.  void foo2(int*);
3.  using namespace std;

4.  void foo2(int *pValue)
5.  {
6.      *pValue = 6;
7.  }
8.
9.  int main()
10. {
11.     int nValue = 5;
12.
13.     cout << "nValue = " << nValue << endl;
14.     foo2(&nValue);
15.     cout << "nValue = " << nValue << endl;
16.     return 0;
17. }
```

Passing by address means passing the address of the argument variable. The function parameter must be a pointer. The function can then dereference the pointer to access or change the value being pointed to.

1. It allows us to have the function change the value of the argument.
2. Because a copy of the argument is not made, it is fast, even when used with large structures or classes.
3. Multiple values can be returned from a function.

Passing Arrays

- Passing 1D array.
  1. int ProcessValues (int a[], int size); // works with ANY 1D array. The compiler only needs to know that the parameter is an array; it doesn't need to know its size. Moreover, ProcessValues() receives access to the actual array, not a copy of the values in the array. Thus any changes made to the array within the function change the original array.
  2. When declaring parameters to functions, declaring an array variable without a size is equivalent to declaring a pointer. Thus "int ProcessValues (int *a, int size)" is equivalent to the above declaration.
- When we pass a 2D array to a function we must specify the number of columns -- the number of rows is irrelevant. We can do: f(int a[][35]) {.....}

Returning Address of Dynamic Memory

- The address of dynamically allocated memory can be returned from argument of function.

```
1.  #include<iostream>
2.  void alloc_double_array(int, double**);
3.  using namespace std;
4.  // Public/dyn_array_from_func.cpp
5.  void alloc_double_array(int size,  double **ppValue)
6.  {
7.      double  *vec;
8.      vec = new double[size];
9.      *ppValue = vec;
10. }
11.
12. int main()
13. {
14.    double *pvec;
15.     alloc_double_array(10,&pvec);
16.    return 0;
17. }
```

3. Pass by reference //Public/Func_arguments/pass_by_ref.cpp

```
1.  #include<iostream>
2.  void foo3(int&);
3.  using namespace std;

4.  void foo3(int &y) // y is now a reference
5.  {
6.      cout << "y = " << y << endl;
7.      y = 6;
8.      cout << "y = " << y << endl;
9.  } // y is destroyed here
10.
11. int main()
12. {
13.     int x = 5;
14.     cout << "x = " << x << endl;
15.     foo3(x);
16.     cout << "x = " << x << endl;
17.     return 0;
18. }
```

Since a reference to a variable is treated exactly the same as the variable itself, any changes made to the reference are passed through to the argument.

```
1.  #include <iostream>
2.  int nFive = 5;
3.  int nSix = 6;
4.  void SetToSix(int *pTempPtr);
5.  using namespace std;
6.
7.  int main()
8.  {
9.     int *pPtr = &nFive;
10.    cout << *pPtr;
11.
12.    SetToSix(pPtr);
13.    cout << *pPtr;
14.    return 0;
15. }
16.
17. // pTempPtr copies the value of pPtr! I.e., pTempPtr stores  the content of pPtr
18. void SetToSix(int *pTempPtr)
19. {
20.     pTempPtr = &nSix;
21.
22.    cout << *pTempPtr;
23. }
```

- **A string reverser program** //~zxu2/Public/wrong_string_reverse.c

```
#include <stdio.h>
/* WRONG! */
char* make_reverse(char *str)
{
    int i, j;
    unsigned int len;
    char newstr[100];
    len = strlen(str) - 1;
    j=0;
    for (i=len; i>=0; i--){
        newstr[j] = str[i];
        j++;
    }
    return newstr; /* now return a pointer to this new string */
}

int main()
{
   char input_str[100];
    char *c_ptr;
    printf("Input a string\n");
    gets(input_str); /* should check return value */
    c_ptr = make_reverse(input_str);
     printf("String was %s\n", input_str);
     printf("Reversed string is %s\n", c_ptr);
}
```

1. The memory allocated for **newstr** when it was declared as an `automatic' variable in make_reverse isn't permanent. It only lasts as long as make_reverse() takes to execute.
2. The newly created array of characters, **newstr**, isn't terminated with a zero character, `\0', so trying to print the characters out as a string may be disastrous.

- **Another string reverser program** //~zxu2/Public/ok_string_reverse.c

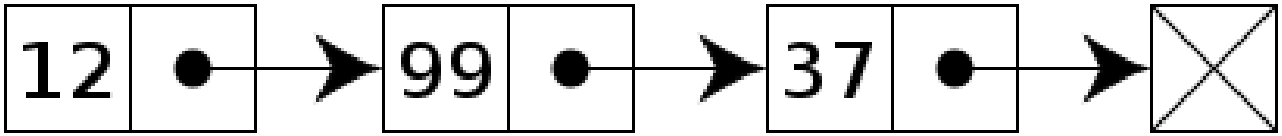
```
#include <stdio.h>
#include <stdlib.h>
char* make_reverse(char *str)
{
    int i;
    unsigned int len;
    char *ret_str, *c_ptr;
    len = strlen(str);
    ret_str = (char*) malloc(len +1); /* Create enough space for the string AND the final \0. */
    c_ptr = ret_str + len; /* Point c_ptr to where the final '\0' goes and put it in */
    *c_ptr = '\0';
/* now copy characters from str into the newly created space. The str pointer will be advanced a char at a time, the cptr pointer will be decremented a char at a time. */
    while(*str !=0){ /* while str isn't pointing to the last '\0' */
        c_ptr--;
       *c_ptr = *str;
        str++; /* increment the pointer so that it points to each  character in turn. */
    }
    return ret_str;
}
int main()
{
     char input_str[100];
     char *c_ptr;
     printf("Input a string\n");
     gets(input_str); /* Should check return value */
     c_ptr = make_reverse(input_str);
     printf("String was %s\n", input_str);
     printf("Reversed string is %s\n", c_ptr);
}
```

The **malloc**'ed space will be preserved until it is explicitly freed (in this case by doing `**free**(c_ptr)'). Note that the pointer to the malloc'ed space is the only way you have to access that memory: lose it and the memory will be inaccessible. It will only be freed when the program finishes.

Singly Linked Lists

- Overall Structure of Singly-Linked Lists

  A list element(node) contains the data plus pointers to the next list items.



- A generic singly linked list node:

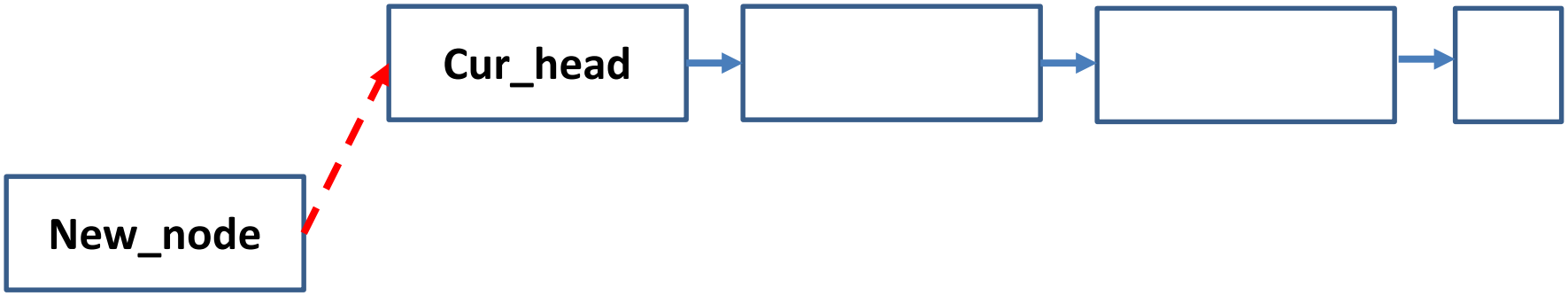
```
struct sl_node {
 int data;
 struct sl_node* next;  // that points to the next node in the list
};
```

Remark.

1. Regarding “struct sl_node* next”, it is okay as it is only a pointer to the incomplete type.
2. Memory for saving a node of singly linked list can be allocated dynamically:

```
// C version
struct sl_node* head = (node*) malloc(sizeof(struct sl_node));
//C++ version
struct sl_node* head = new (struct sl_node);
```

- Inserting to a singly linked list from the beginning



Following codes are needed:

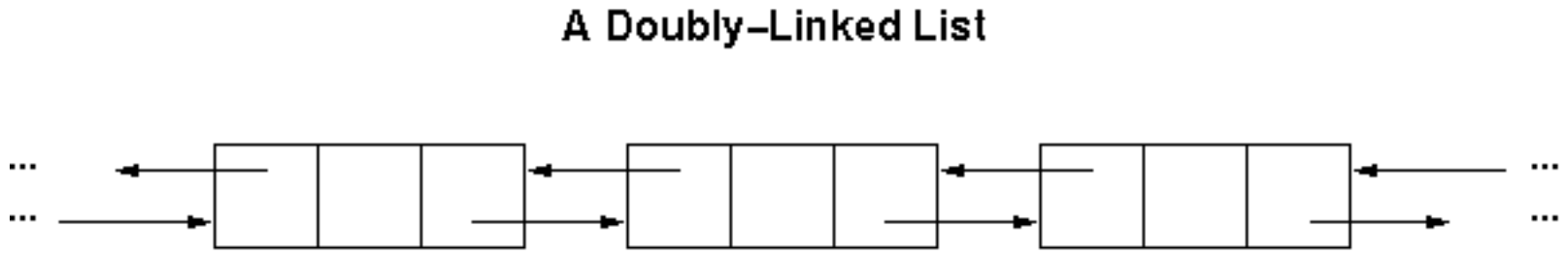
1. New_node->next = Cur_head;
2. Cur_head = New_node;

// See Public/dyn_linked_list.cpp

Doubly Linked Lists

- Overall Structure of Doubly Linked Lists

  A list element contains the data plus pointers to the next and previous list items.



- A generic doubly linked list node:

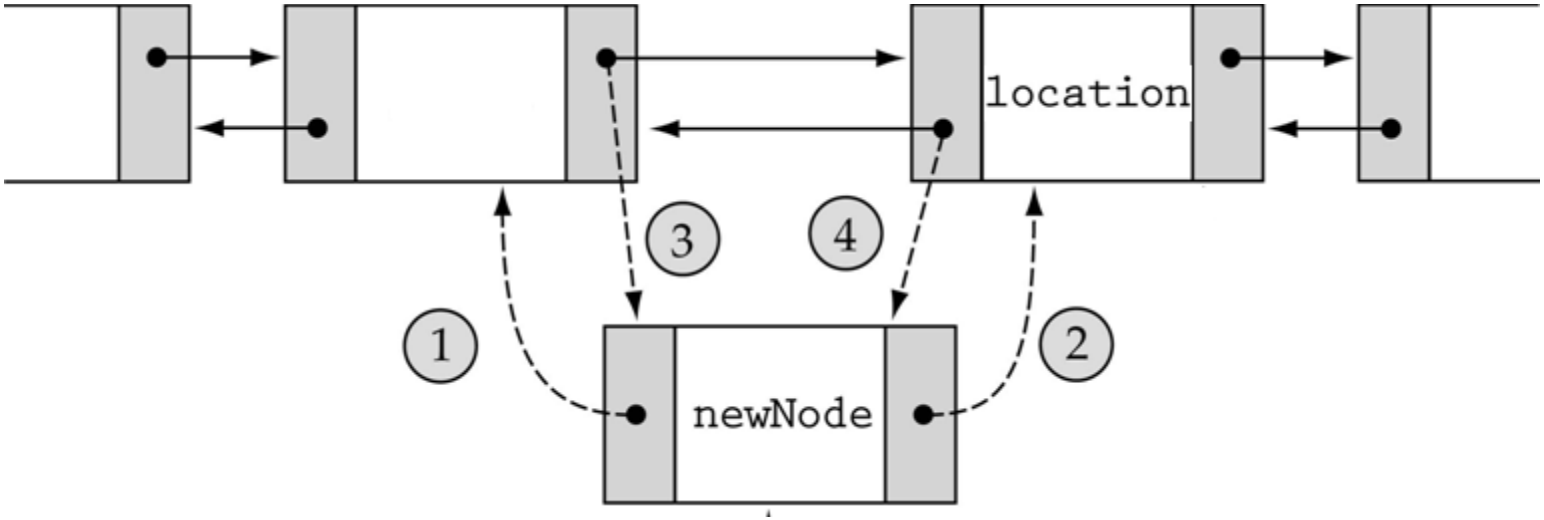
```
struct node {
 int data;
 struct node* next;  // that points to the next node in the list
 struct node* prev; // that points to the previous node in the list.
};
```

Remark.

1. Regarding “struct node* next” and “struct node* prev”, they are okay as they are only pointers to the incomplete type.
2. Memory for saving a node can be allocated dynamically:

```
// C version
struct node* head = (node*) malloc(sizeof(struct node));
//C++ version
struct node* head = new (struct node);
```

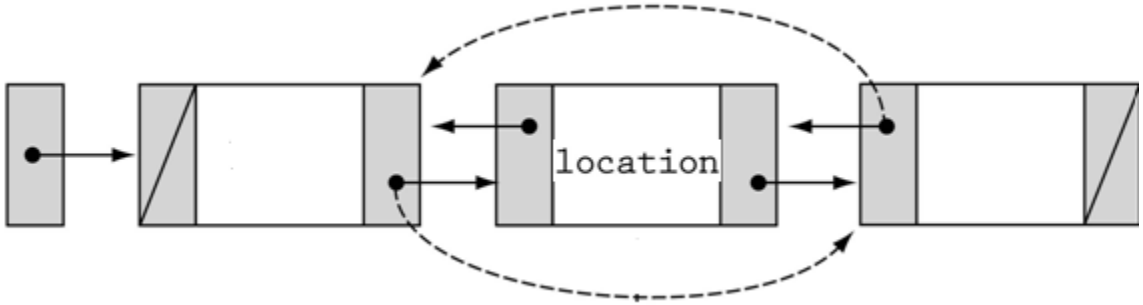
- Inserting to a Doubly Linked List



Following codes are needed:

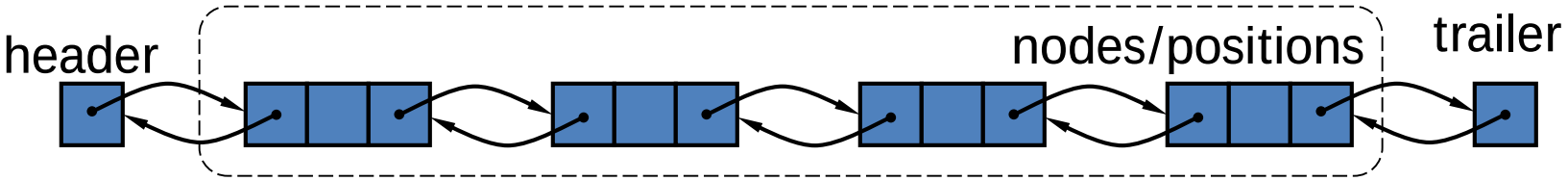
1. newNode->prev = location->prev;
2. newNode->next = location;
3. location->prev->next=newNode;
4. location->prev = newNode;

- Deleting “location” node from a Doubly Linked List

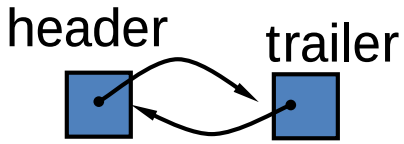



```
node* temp;
1.  temp = location->prev;
2. temp->next =location->next;
3. (temp->next)->prev = temp;
4. free(location);
```

- Special trailer and header nodes and initiating doubly linked list



1. To simplify programming, two special nodes have been added at both ends of the doubly-linked list.
2. Head and tail are dummy nodes, and do not store any data elements.
3. Head: it has a null-prev reference (link).
4. Tail: it has a null-next reference (link).




```
//Initialization:
node header, trailer;
1. header.next = &trailer;
2. trailer.prev = &header;
```

- Insertion into a Doubly Linked List from the End

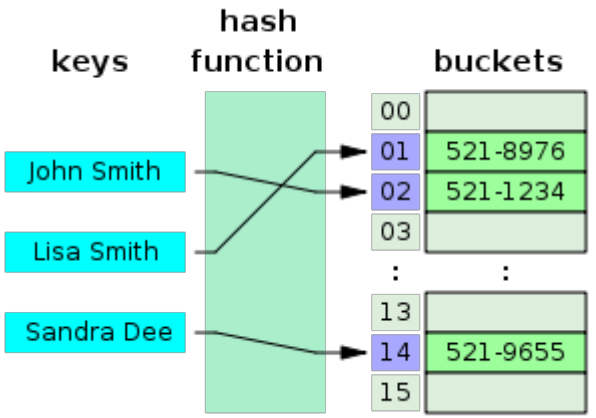
AddLast algorithm – to add a new node as the last of list:

```
void addLast( node *T,  node *trailer)
{
        T->prev = trailer->prev;
        trailer->prev->next = T;
        trailer->prev = T;
        trailer->prev->next = trailer;
}
```

Hash Table

- A hash is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the correct value can be found.

  See also http://xlinux.nist.gov/dads/HTML/hashtab.html



Hashing: Given a key, the algorithm computes an index that suggests where the entry can be found.

index = f(key, array_size);

Remark: 1. see ANSI C for Programmers on UNIX Systems by Tim Love
2. C++ STL has its implementation

C++ Class

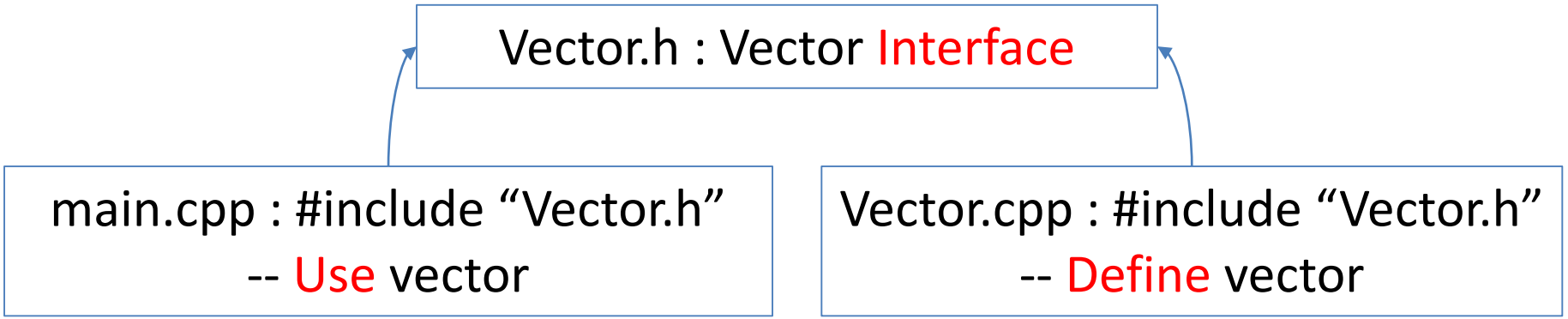
- A class is a user-defined type provided to represent a concept in the code of a program. It contains data and function members.

```
// Vector.h // see ~zxu2/Public/C++_sample_vec

#if !defined(_VECTOR_H)
#define _VECTOR_H

class Vector{
   private:
            double* elem; // elem points to an array of sz doubles
            int         sz;
   public:
            Vector(int s); // constructor: acquire resources
            ~Vector(){delete[] elem;}                    //destructor : release resources
            double& operator[](int);    //operator overloading
            int size() const;           //const indicates that this function does not modify data
};
#endif /* !defined(_VECTOR_H) */
```

- A class is defined to have a set of *member*s, which can be data, function or type members.
- The **public** keyword specifies that those members are accessible from any function.
- The **private** keyword specifies that those members are accessible only from member functions and friends of the class.
- The interface is defined by **public** members of a class, and **private** members are accessible only through that interface.
- A “function” with the same name as its class is called a *constructor*, which is used to construct objects of a class. A constructor is guaranteed to be used to initialize objects of its class.

```
// Vector.cpp, here we define interfaces to the data
#include “Vector.h”
Vector::Vector(int s):elem{new double[s]}, sz{s} // constructor: acquire resources
{
        for(int I = 0; I < s; I++)   elem[I] = 0;
}
// :elem{new double[s]}, sz{s} is the member initializer list.

double& Vector::operator[](int i)  //overloading operator
{
          return elem[i];
}
// Access to elements is provided by a subscript function, operator[].
int Vector::size() const
{
    return sz;
}
```

```
// main.cpp. To compile icpc main.cpp Vector.cpp
#include “Vector.h”
#include <iostream>

int main()
{
    Vector v(10);
    v[4] = 2.0;
    std::cout<<“size of vector = ”<<v.size() <<std::endl;
}
```



Modularity

- A C++ program consists of many separately developed parts, such as functions, user-defined types and templates.
- The key to managing a program is to clearly define interactions among these parts.
- The first step is to distinguish between interface to a part and its implementation.
- C++ represents interfaces by *declarations*. They are usually in *.h files. For example:

```
double sqrt(double); // the square root
class Vector{
public:
        Vector(int s);
};
```

- Function *definitions* (bodies) are elsewhere. They are usually in *.cpp files.

Enumerations

- Enumerations are used to represent small set of “integer” values. They are used to make code more readable and less error-prone.
- User-defined type.

```
enum class Color {red, blue, green};
enum class Traffic_light {green, yellow, red};

// class after enum specifies that an enumeration is strongly typed and that its
//  enumerators are scoped.
Color x = red;                    // error: which red
Color y = Traffic_light:: red; // error: that red is not a Color
Color z = Color::red;             // OK.
int   I = Color::red;             // error: Color::red is not integer
```

Namespaces

- A mechanism for expressing that some declarations belong together and that their names shouldn't clash with other names.

```
namespace My_code{
    class complex {};
    complex sqrt(complex);
    …
    int main();
}

int My_code::main()
{
     complex z(1,2);
     //…
}

int main()
{
     return My_code::main();
}
```

Error Handling

- **Exceptions report errors found at run time. Exceptions are used to signal errors that cannot be handled locally, such as the failure to acquire a resource in a constructor.**
- Three related keywords**: throw, try, catch**

```
double Vector::operator[](int i)
{
    if(i<0 || size()<=i) throw out_of_range{“operator[]”};
     return elem[i];
}
//throw transfers control to a handler for exceptions of
//type out_of_range defined in the standard library (in <stdexcept>);
void f(Vector& v)
{
   //…
   try{// exceptions here are handled by the handler defined below
        v.[v.size()] = 7;  // try to access beyond the end of v
   }
   catch (out_of_range){ // … handle range error
   }
}
// put the code for which we are interested in handling exceptions
// into a try-block.
// See: https://isocpp.org/wiki/faq/exceptions
```

Friends

An ordinary member function declaration specifies three things:

1) The function can access the private part of the class.
2) The function is in the scope of the class.
3) The function must be invoked on an object (has a this pointer).

By declaring a nonmember function a friend, we can give it the first property only.

Example. Consider to do multiplication of a **Matrix** by a **Vector**. However, the multiplication routine cannot be a member of both. Also we do not want to provide low-level access functions to allow user to both read and write the complete representation of both Matrix and Vector. To avoid this, we declare the **operator*** a **friend** of both.

```
class Matrix;

class Vector{
    double *v;
    friend Vector operator*(const Matrix&, const Vector&);
};
class  Matrix{
     double **elem;
     friend Vector operator*(const Matrix&, const Vector&);
};
// Now operator*() can reach into the implementation of both Vector and Matrix.
Vector* operator*(const Matrix& m, const Vector& v)
{
            Vector *r=new Vector(Mat.size_row());
            for(int I = 0; I< 4; I++)
            {
                  (*r)[I]=0;
                  for(int J = 0; J< 4; J++)
                  {
                        (*r)[I] +=m(I,J)*v[J];
                  }
            }
             return r;
}
```

- Check ~zxu2/Public/C++_mat_vec_multi for an implementation which uses dynamic memory allocation.

Copy constructors

```
//Consider the example at zxu2/Public/C++_sample_vec_2
// what happens to the following piece of code.

int main()
{
    Vector v(10);
    Vector v2 = v;
    int i;
    v[4] = 2.0;
}

//output of run:
*** glibc detected *** ./vec: double free or corruption (fasttop):
0x0000000001023010 ***
======= Backtrace: =========
```

- First, if a copy constructor "X(const X&)" is not declared, the compiler gives one implicitly. The implicit copy constructor does a member-wise copy of the source object.
- For example:

```
class MyClass { public:
    int x;
    char c;
    std::string s;
 };

//the compiler-provided copy constructor is exactly equivalent to:
MyClass::MyClass( const MyClass& other ) :
   x( other.x ), c( other.c ), s( other.s ) {}

main()
{
   Myclass my1;
   my1.x = 2;
   Myclass my2 = my1;  // the member x of my2 is 2 now.
}
```

- The most common reason the default copy constructor is not sufficient is because the object contains raw pointers and a "deep" copy of the pointer is needed. That is, instead of copying the pointer itself, copy what the pointer points to.

- An overloading operator= can solve the previous problem in ~zxu2/Public/C++_sample_vec_2.

```
class Vector{
    …
    Vector& operator=(const Vector&); // assignment
};

Vector& Vector::operator=(const Vector&  a)
{
   if(this != &a)
     delete [] elem;
   elem = new double [a.size()];
   sz = a.size();
   for(int I = 0; I < sz; I++)
      elem[I] = a.elem[I] ;

   return *this;
}
```

```
// the previous overloaded "=" make the following code safe
main(){
   Vector v1(10), v2(20);
    v1 = v2;
}
```

```
// the previous overloaded "=" does not make the following code safe
main(){
   Vector v1(10);
   Vector v2 = v1;  // we need initialization here, not assignment.
}
// A copy constructor is  needed to resolve this situation.
```

```
class Vector{
     Vector(int s); // constructor: acquire resources and create objects
     Vector& operator=(const Vector&); // assignment
     Vector(const Vector&); // copy constructor. It takes care of initialization
                            // by an object of the same type Vector.
};

Vector& Vector::operator=(const Vector&  a){
   if(this != &a)
      delete elem;
   elem = new double [a.size()];
   sz = a.size();
   for(int I = 0; I < sz; I++)      elem[I] = a.elem[I] ;
   return *this;
}

Vector::Vector(const Vector&  a) {
   elem = new double [a.size()];
   sz = a.size();
   for(int I = 0; I < sz; I++)
      elem[I] = a.elem[I] ;
}
```

```
// Now this is safe code with previous overloaded
// "=" and copy constructor
int main()
{
    Vector v(3);
    v[1] = 2.0;
     Vector v2= v;  // this is initialization
}
```

Abstraction Mechanisms

- C++ supports for abstraction and resource management without going into details.
- *Concrete classes, abstract classes, class hierarchies*.
- Templates are introduced as a mechanism for parameterizing types and algorithms with (other) types and algorithms.

Concrete Types

- Basic idea of concrete classes is that they behave like build-in types. Representations of concrete classes are part of definition of the class.
- Concrete classes allow to:
  1. Place objects of concrete classes on the stack, and in other objects.
  2. Refer to objects directly (and not just through pointers and references).
  3. Initialize objects immediately and completely; and copy objects.

- **Example. An Arithmetic Type**

```
class complex{
   double re, im;  //represention: two doubles
Public:
   complex(double r, double i) :re{r}, im{i}{}
   complex() : re{0}, im{0}{} //default constructor
   double real() const {return re;}
   double image() const {return im;}
   complex& operator +=(complex z){
       re+=z.re;  im +=z.im;  return *this;
   } // add to re and im and return the result
};
// const specifier on the functions indicate that these functions
// do not modify the object for which they are called.
// multiple constructors are allowed.
```

- **Example. A container**
- A *container* is an object holding a collection of elements.

```
class Vector{
    private:
        double* elem;
        int         sz;
    public:
        Vector(int s);
        ~Vector(){delete[] elem;}
       double& operator[](int);
         int size() const;
};
```

Abstract Types

- An abstract type is a type that decouple interface and representation.
- Since the representation of an abstract type is not known, objects must be allocated on the free store and access them through references or pointers.
- A concrete type can be derived from an abstract type.

```
class Container {
public:
   virtual double& operator[](int) =0; // pure virtual function
   virtual  int size() const = 0;              //  const member function
   virtual ~Container(){}
};
// a function declared “virtual” is called a virtual function. A class derived from
//  Container provides an implementation for the container interface.
// “=0” syntax says that the function is pure virtual. A class derived from Container
// must define the function.
// A class with pure virtual function is called an abstract class.
```

Virtual function is an inheritable and overriddable function, and gives programmer capability to call member function of different class by a same function call depending upon different context.

- A container (concrete type) that implements the functions required by the interface defined by the abstract class **Container** could use the concrete class **Vector**.

```
class Vector_container: public Container {
    Vector v;
public:
    Vector_container(int s):v(s){}   // Vector of s elements
    ~Vector_container(){}
    double & operator[](int i) {return v[i];}
    int size() const {return v.size(); }
};
 // : public can be read as “is derived from”. Vector_container is derived
// from class Container. Container is said to be a base of class Vector_container.
// members operator[] and size() are said to override
//  the corresponding members in the base.
```

C++ Inheritance

- Any class type may be declared as derived from one or more base classes which, in turn, may be derived from their own base classes, forming an inheritance hierarchy.
- Syntax: class derived-class: access-specifier base-class

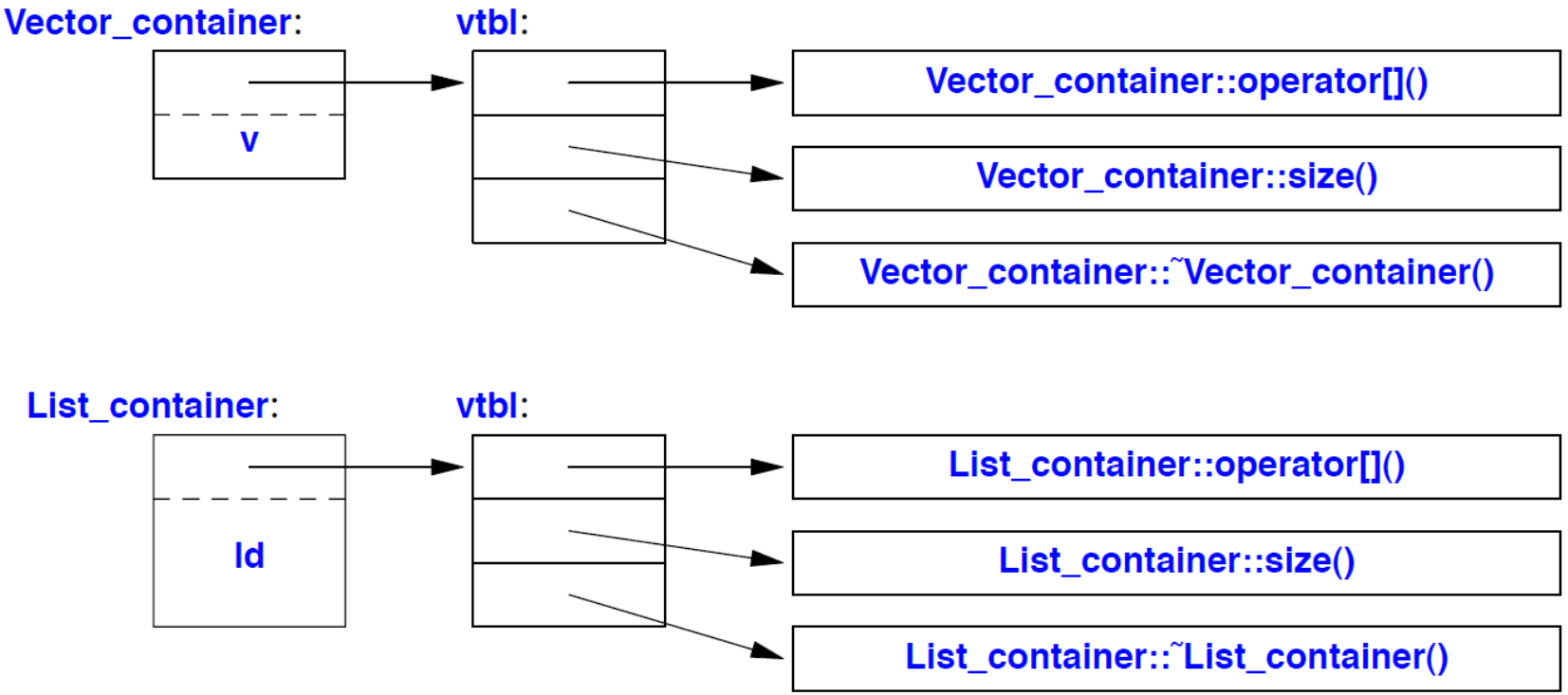
```
// Base class
class Shape {
  public:
    void setWidth(int w)    {       width = w;     }
    void setHeight(int h)    {       height = h;    }
  protected:    // derived class can access the protected data in base class.
    int width;
    int height;
};

// Derived class
class Rectangle: public Shape{
  public:
    int getArea()     {       return (width * height);      }
};
```

```
// Container can be used like:
void use(Container& c){
    const int sz = c.size();
    for(int i=0; i<sz; i++)
        cout<<c[i]<<"\n";
}
void g(){
    Vector_container   vc(10); //ten elements
    use(vc);
}
```

```
class List_container: public Container {
    std::list<double> ld;  // (standard-library) list of doubles
public:
    List_container(initializer_list<double> s):ld(s){}
    ~List_container(){}
    double & operator[](int i);
    int size() const {return v.size(); }
};

double& List_container::operator[](int i){
   for(auto&x : ld){ //read as: for each x in ld. Range-based for loop introduced since C++11
      if(i==0)return x; --i;
   }
}
void h(){
  List_container lc={1, 2, 3, 4, 5};
  use(lc);
}
```

Virtual Functions

- When h() calls use(), List_container's operator[]() must be called. When g() calls use(), Vector_container's operator[]() must be called. How to resolve?
- Vtbl: virtual function table
- Compiler converts name of virtual functions into an index into a table (vtbl) of pointers to functions. Each class with virtual functions has its own vtbl.



```
/* Example to demonstrate the working of virtual function in C++ programming. */
#include <iostream>
using namespace std;
class B {
   public:
    virtual void display()     /* Virtual function */     { cout<<"Content of base class.\n"; }
};
class D1 : public B {
   public:
     void display()       { cout<<"Content of first derived class.\n"; }
};
class D2 : public B{
   public:
     void display()       { cout<<"Content of second derived class.\n"; }
};

int main()
{
   B *b;      D1 d1;       D2 d2;

/* b->display();  // You cannot use this code here because the function of base class is virtual. */
   b = &d1;
   b->display();   /* calls display() of class derived D1 */
   b = &d2;
   b->display();   /* calls display() of class derived D2 */
   return 0;
}
```

Operator Overloading

Overloadable operators

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

```
// complex.h  //see ~zxu2/Public/complex_class
class complex{
private:
   double real, image;
public:
   complex operator+(const complex&);
   complex& operator+=(complex);
   complex& operator=(const complex&);
   complex(double a, double b)    {       real = a;        image = b;    };
};
```

Remark:

A binary operator (e.g. a+b, a-b, a*b) can be defined by either a non-static member function taking one argument or a nonmember function taking two arguments. For any binary operators @, aa@bb is aa.operator@(bb), or operator@(aa,bb).

A unary operator can be defined by either a non-static member function taking no arguments or a nonmember function taking one argument. For any prefix unary operator (e.g. –x, &(y)) @, @aa can be interpreted as either aa.operator@() or operator@(aa). For any post unary operator (e.g. a--) @, aa@ can be interpreted as either aa.operator@(int) or operator@(aa,int).

A non-static member function is a function that is declared in a member specification of a class without a static or friend specifier.

- Operators [], (), ->, ++, --, new, delete are special operators.

```
struct Assoc{
   vector<pair<string,int>> vec;   // vector of (name, value) pairs
   int& operator[](const string&);
};

int& Assoc::operator[](const string& s)
{
   for(auto x:vec) if(s == x.first) return x.second;
    vec.push_back({s,0});          // initial value: 0
    return vec.back().second;   // return last element.
}

int main()
{
    Assoc values;
    string buf;
    while(cin>>buf) ++values[buf];
     for(auto x: values.vec) cout<<‘{‘ <<x.first <<‘,’<<x.second <<“}\n”;
}
```

C++ Template

- C++ templates (or parameterized types) enable users to define a family of functions or classes that can operate on different types of information. See also http://www.cplusplus.com/doc/oldtutorial/templates/
- Templates provides direct support for generic programming.

```
// min for ints
int min_i( int a, int b ) {
  return ( a < b ) ? a : b;
}

// min for longs
long min_l( long a, long b ) {
  return ( a < b ) ? a : b;
}

// min for chars
char min_c( char a, char b ) {
  return ( a < b ) ? a : b;
}
```

```
//a single function template implementation
template <typename T> T min( T a, T b ) {
  return ( a < b ) ? a : b;
}

int main()
{
   min<double>(2, 3.0);
}
```

See Public/ACMS40212/C++_template_function

Class template

```
// declare template
template<typename C> class String{
private:
        static const int short_max = 15;
        int sz;
        char *ptr;
        union{
                int space;
                C ch[short_max+1];
        };
public:
        String ();
       C& operator [](int n) {return ptr[n]};
        String& operator +=(C c);
};
```

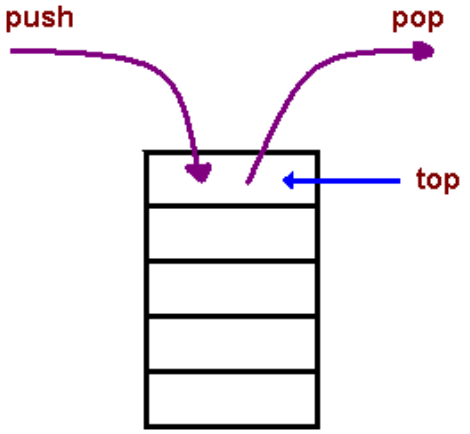
```
// define template
Template<typename C>
String<C>::String()       //String<C>'s constructor
:sz{0},ptr{ch}
{
   ch[0]={};
}
Template<typename C>
C& String<C>::operator+=(C c)
{
// … add c to the end of this string
     return *this;
}
```

Remark: keyword this is a pointer to the object for which the function was invoked. In a non-const member function of class X, the type of this is X*.

```
// template instantiation
…
String<char> cs;
String<unsigned char> us;
Struct Jchar{…};  //Japanese character
String <Jchar> js;
```

Stacks

- A stack is a container of objects that are inserted and removed according to the last-in first-out (**LIFO**) principle. In the pushdown stacks only two operations are allowed: **push** the item into the stack, and **pop** the item out of the stack.




```
template <typename T>
class stack {
    T*    v;
    T*    p;
    int    sz;

public:
   stack (int s)       {v = p = new T[sz = s];}
   ~stack()            {delete[] v;}
   void push (T a)  { *p = a;  p++;}
   T pop()             {return *--p;}
   int  size()     const {return p-v;}
};

stack <char> sc(200);    // stack of characters
```

Remark:
The template <typename T> prefix specifies that a template is being declared and that an argument T of type type will be used in the declaration. After its introduction, T is used exactly like other type names. The scope of T extends to the end of the declaration that template <typename T> prefixes.

Non template version of stack of characteristics

```
class stack_char {
    char*    v;
    char*    p;
    int    sz;


public:
   stack_char (int s)       {v = p = new char[sz = s];}
   ~stack_char()            {delete[] v;}
   void push (char a)  { *p = a;  p++;}
   char pop()              {return *--p;}
   int  size()     const {return p-v;}
};


stack_char sc(200);    // stack of characters
```

C++ STL

- STL consists of the iterator, container, algorithm and function object parts of the standard library.
- A container holds a sequence of objects.
  - *Sequence container*:

vector<T,A> // a contiguously allocated sequence of Ts

list<T,A> //a doubly-linked list of T

forward_list<T,A> // singly-linked list of T

Remark: A template argument is the allocator that the container uses to acquire and release memory

  - *Associative container:*

map<K,V,C,A> // an ordered map from K to V. Implemented as binary tree

unordered_map<K,V,H,E,A> // an unordered map from K to V
// implemented as hash tables with linked overflow

Container adaptor:

queue<T,C> //Queue of Ts with push() and pop()

stack<T,C> //Stack of Ts with push() and pop()

  - *Almost container:*

array<T,N> // a fixed-size array N contiguous Ts.

string

```cpp
#include <iostream>
#include <vector>
using namespace std;
int main()
{ // create a vector to store int
    vector<int> vec; int i;
    // display the original size of vec
     cout << "vector size = " << vec.size() << endl;
     // push 5 values into the vector
    for(i = 0; i < 5; i++){
        vec.push_back(i);
    }
    // display extended size of vec
    cout << "extended vector size = " << vec.size() << endl;
    // access 5 values from the vector
    for(i = 0; i < 5; i++){
        cout << "value of vec [" << i << "] = " << vec[i] << endl;
    }
    // use iterator to access the values
    vector<int>::iterator v = vec.begin();
    while( v != vec.end()) {
        cout << "value of v = " << *v << endl; v++;
    }
    vec.erase(vec.begin()+2); // delete the 3rd element in the vec.
    return 0;
}  // http://www.tutorialspoint.com/cplusplus/cpp_stl_tutorial.htm
```

STL Iterators

An iterator is akin to a pointer in that it provides operations for indirect access and for moving to point to a new element. A *sequence* is defined by a pair of iterators defining a half-open range [begin:end), i.e., never read from or write to *end.

Iterator can be incremented with ++, dereferenced with *, and compared against another iterator with !=.

```
// look for x in v
auto p = find(v.begin(),v.end(),x);
if(p != v.end()){
   // x found at p
}
else {
    // x not found in [v.begin():v.end())
}
```

```
// use iterator to access the values
vector<int>::iterator v = vec.begin();
while( v != vec.end()) {
  cout << "value of v = " << *v << endl;
  v++;
}
```

- **Operators**
  - **Operator *** returns the element of the current position.
  - **Operator ++** lets the iterator step forward to the next element.
  - **Operator ==** and **!=** returns whether two iterators represent the same position
  - **Operator =** assigns an iterator.
- **begin()** returns an iterator that represents the beginning of the element in the container
- **end()** returns an iterator that represents the position behind the last element.
- *container::iterator* is provided to iterate over elements in read/write mode
- *container::const_iterator* in read-only mode
- *container::iterator{first}* of (unordered) maps and multimaps yields the second part of key/value pair.
- *container::iterator{second}* of (unordered) maps and multimaps yields the key.

```
//Public/ACMS40212/C++_STL_samples/map_by_hash.cpp. Use intel icc ver14 to compile
#include <unordered_map>
#include <iostream>
#include <string>
using namespace std;
int main ()
{
  std::unordered_map<std::string,double> mymap = {
     {"mom",5.4},     {"dad",6.1}, {"bro",5.9} };

  std::cout << "mymap contains:";
  for ( auto it = mymap.begin(); it != mymap.end(); ++it )
    std::cout << " " << it->first << ":" << it->second;
  std::cout << std::endl;

  std::string input;
  std::cout << "who? ";
  getline (std::cin,input);

  std::unordered_map<std::string,double>::const_iterator got = mymap.find (input);
  if ( got == mymap.end() )
    std::cout << "not found";
  else
    std::cout << got->first << " is " << got->second;
  std::cout << std::endl;
  return 0;
}
```

```
//Public/ACMS40212/C++_basics/map_by_tree.cpp
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
  map<string, string> mascots;
  mascots["Illinois"]      = "Fighting Illini";   mascots["Indiana"]       = "Hoosiers";
  mascots["Iowa"]          = "Hawkeyes";  mascots["Michigan"]      = "Wolverines";
  mascots["Michigan State"] = "Spartans";   mascots["Minnesota"]     = "Golden Gophers";
  mascots["Northwestern"]   = "Wildcats";    mascots["Ohio State"]     = "Buckeyes";
  mascots["Penn State"]     = "Nittany Lions";    mascots["Purdue"]        = "Boilermakers";
  mascots["Wisconsin"]      = "Badgers";
  for (;;)
  {
    cout << "\nTo look up a Big-10 mascot, enter the name "  << "\n of a Big-10 school ('q' to quit): ";
    string university;
    getline(cin, university);
    if (university == "q") break;
    map<string, string>::iterator it = mascots.find(university);
    if (it != mascots.end()) cout << "--> " << mascots[university] <<  endl;
    else
      cout << university << " is not a Big-10 school " << "(or is misspelled, not capitalized, etc?)" << endl;
  }
}
```

- Using template to implement Matrix.
- See zxu2/Public/ACMS40212/C++template_matrix

  driver_Mat.cpp Matrix.cpp Matrix.h

Modularity

- One way to design and implement the structured program is to put relevant data type together to form aggregates.
- Clearly define interactions among parts of the program such as functions, user-defined types and class hierarchies.
- Try to avoid using nonlocal variables.
- At language level, clearly distinguish between the interface (declaration) to a part and its implementation (definition).
  - Use header files to clarify modules
  - See ~zxu2/Public/C++_sample_vec
- Use separate compilation
  - Makefile can do this
- Error handling
  - Let the return value of function be meaningful.
    - See ~zxu2/Public/dyn_array.c
  - Use Exceptions
    - ~zxu2/Public/C++_sample_vec

Use of Headers

- Use “include guards” to avoid multiple inclusion of same header

```
#ifndef _CALC_ERROR_H
#define _CALC_ERROR_H
…
#endif
```

- Things to be found in headers
  - Include directives and compilation directives

    ```
    #include <iostream>
    #ifdef __cplusplus
    ```

  - Type definitions

    ```
    struct Point {double x, y;};
    class  my_class{};
    ```

  - Template declarations and definitions

    ```
    template template <typename T> class QSMatrix {};
    ```

  - Function declarations

    ```
    extern int my_mem_alloc(double**,int);
    ```

  - Macro, Constant definitions

    ```
    #define VERSION 10
    const double PI = 3.141593 ;
    ```

Multiple Headers

- For large projects, multiple headers are unavoidable.
- We need to:
  - Have a clear logical organization of modules.
  - Each .c or .cpp file has a corresponding .h file. .c or .cpp file specifies definitions of declared types, functions etc.
- See ~zxu2/Public/C++_mat_vec_multi for example.

References:

- Tim Love, ANSI C for Programmers on UNIX Systems
- http://www.tutorialspoint.com/cprogramming
- Bjarne Stroustrup, The C++ Programming Language