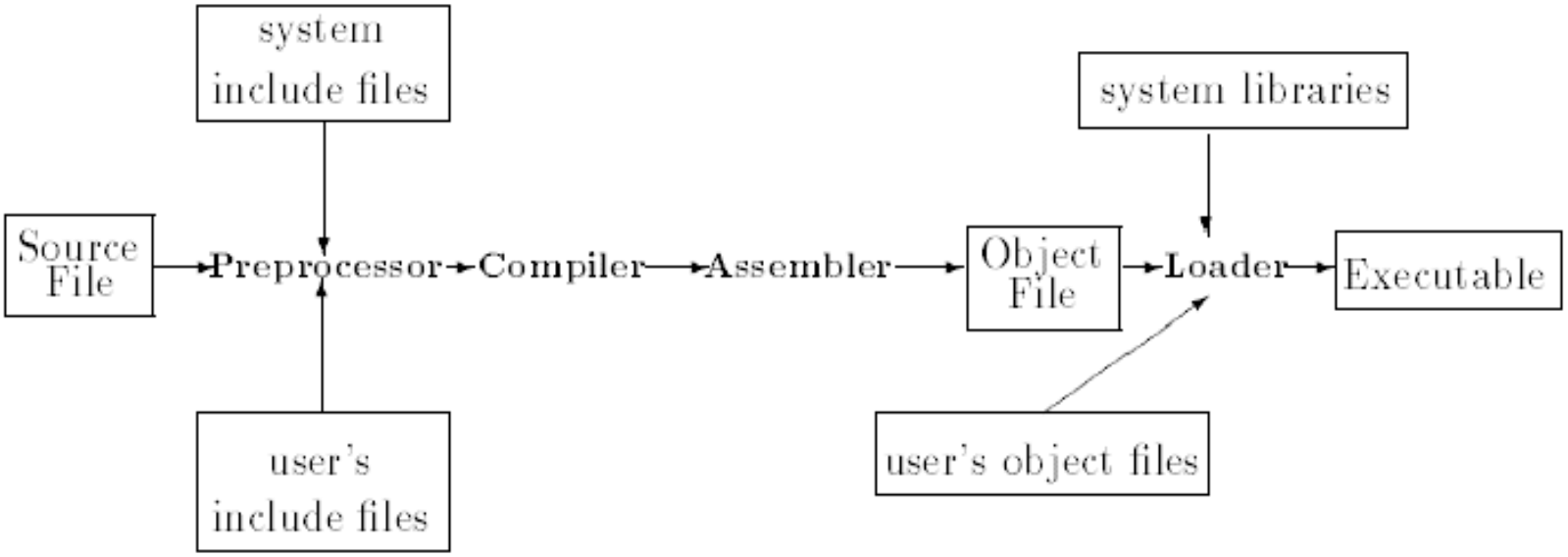
Short Notes on C/C++

Prepared for self use : Jaydeep Shah
radhey04ec@gmail.com

- 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

- 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;
```

Remark: variables of structure type can not be compared.

Do not do:

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

Pointers

- A variable can be viewed as a specific block of memory in the computer memory which can be accessed by the identifier (the name of the variable). How compiler assign memory ??
  - 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 */
```

- Pointers and arrays

```
int  a[100],  *ptr_a;
ptr_a = &(a[0]);   /* or ptr_a = a; */ // Point ptr_a to the first element in a[]
/* now increment ptr_a to point to successive elements */
for(int i =0; i < 100; i++)
{
     printf("*ptr_a is %d\n", *ptr_a);
    ptr_a++;   /*or ptr_a += 1; */ // ptr_a is incremented by the length of an int
                                    // and points to the next integer, a[1], a[2] etc.
}
```

- 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);
}
```

Dynamic Memory Allocation in C/C++

Motivation

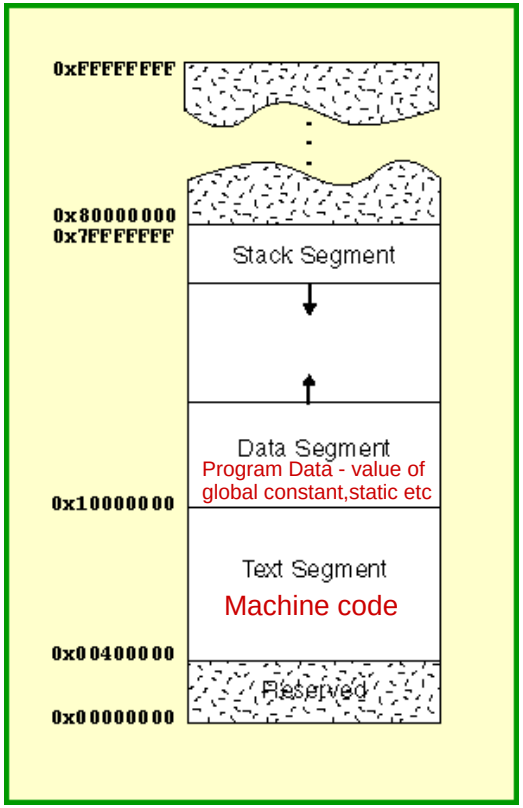
```
 /*   a[100]  vs. *b or *c     */
Func(int array_size)
{
    double  k, a[100], *b, *c;
    b = (double *) malloc(array_size * sizeof(double));  /* allocation in C*/
    c = new double[array_size];  /* allocation in C++ */
    …
}
```

- 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.

icc –g dyn_mem_alloc.cpp

Stack vs Heap



When a program is loaded into memory:

- Machine code is loaded into **text** segment
- **Stack** segment allocate memory for automatic variables within functions
- **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/C++

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.

C++:

- "new" operator
  -- pointer = new type For single variable
  -- pointer = new type [number_of_elements] For memory chunk / Array
  It returns a pointer to the beginning of the new block of memory allocated.
- "delete" operator
  -- delete pointer; For single variable
  -- delete [] pointer; For memory chunk / array

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
    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.

We can not use reference concept for passing array

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;
}
```

- **Illustration**

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;
    }

}
```

Declare as pointer Array - to store memory Address

Important concept
Use of malloc with Array

Release Dynamic Memory

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

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.

3. Pass by reference

```
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.

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.

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 ;
    ```