Pointers

- Pointers
- Pointers and Arrays
- Pointers and function arguments
- Dynamic memory management
- New and delete

Pointers are used to:
- Access array elements
- Passing arguments to functions when the function needs to modify the original argument
- Passing arrays and strings to functions
- Obtaining memory from the system
- Creating data structures such as linked lists
- Many operations that require pointers in C can be carried out without pointers in C++ using reference arguments instead of pointers, strings instead of char arrays or vectors instead of arrays
- Some operations still require pointers, for example creating data structures such as linked lists and binary trees

Each variable in a program occupies a part of the computer’s memory, for example an integer variable occupies 4 bytes of memory.

The location of the piece of memory used to store a variable is called the address of that variable.

An address is some kind of number similar to house numbers in a street that is used to locate the information stored in that particular variable.

int i; address of i

char c; address of c

short s; address of s

int i = 17;
int *ptr; // defines a pointer to an integer variable
ptr = &i; // assign the address of x to the pointer
cout << *ptr << endl; // prints the value of variable i

t *ptr; // pointer to a single character

Address of i

0x1054
0x1055
0x1056
0x1057
0x1058
0x1059
0x1060
1001100

Address of c

0x1054
0x1055
0x1056
0x1057
0x1058
0x1059
0x1060
1001100

Address of s

0x1054
0x1055
0x1056
0x1057
0x1058
0x1059
0x1060
1001100

A pointer variable is a variable that holds address values.

Each data type has its own pointer variable, pointer to int, pointer to double, pointer to char, ...

C/C++ uses the address-of operator & to get the address of an variable.

C/C++ uses the indirection or contents-of operator * to access the value of the variable pointed by.

```cpp
int i = 17;
int *ptr; // defines a pointer to an integer variable
ptr = &i; // assign the address of x to the pointer
cout << *ptr << endl; // prints contents of variable i
```
**Pointer Variables**

```c
int i;
int *ptr;
ptr=&i;
cout << *ptr << endl;
```

**Pointers and Arrays**

- There is a close association between pointers and arrays
- Arrays can be accessed using pointers
- The name of an array is also a constant pointer to the data type of the elements stored in the array

```c
int array[5] = { 23, 5, 12, 34, 17 }; // array of 5 ints
for (int i=0; i< 5; i++)
    cout << array[i] << endl;  // using index to access elements
for (int i=0; i< 5; i++)
    cout << *(array+i) << endl;  // using pointer to access elements
// array is of type pointer to integer
```

**Pointers as Function Arguments**

- C/C++ offers three different ways to pass arguments to a function
  - by value : void f(int x);
  - by reference : void f(int& x);
  - by pointer : void f(int* x);
- In passing by value the function obtains only a local copy of the variable, so that changes to the local variable have no impact on the argument with which the function was invoked
- In passing by reference and passing by pointer the function manipulates the original variable rather than only a copy of it

```c
int v;  // defines variable v of type int
int w;  // defines variable w of type int
int *p; // defines variable p of type pointer to int
p=&v; // assigns address of v to pointer p
v=3; // assigns value 3 to v
*p=7; // assigns value 7 to v
p=&w; // assigns address of w to pointer p
*p=12; // assigns value 12 to w
```

- Using the indirection operator *p to access the contents of a variable is called indirect addressing or dereferencing the pointer

```c
int v;  // defines variable v of type int
int w;  // defines variable w of type int
int *p; // defines variable p of type pointer to int
p=&v; // assigns address of v to pointer p
v=3; // assigns value 3 to v
*p=7; // assigns value 7 to v
p=&w; // assigns address of w to pointer p
*p=12; // assigns value 12 to w
```
Pointers as Function Arguments

```c
void swap(double & x, double & y) {
    double tmp=x;
    x=y;   // access variable by its alias name
    y=tmp;
}
void swap(double * ptr1, double * ptr2) {
    double tmp=*ptr1;
    *ptr1=*ptr2;  // de-referencing pointer
    *ptr2=tmp;
}
double a=3.0;
double b=5.0
swap(a,b);   // call by reference to variables a and b
swap(&a, &b); // call by pointer using the addresses of a and b
```

BubbleSort

```c
void bsort(double *ptr, int n) {   // pass pointer to array and
    // size of array as arguments to bsort
    int j,k; // indices to array
    for(j=0; j<n-1; j++)    // outer loop
        for(k=j+1; k<n; k++)   // inner loop starts at outer
            if(*(ptr+j) > *(ptr+k))
                swap(ptr+j,ptr+k);
}
double array[6] = { 2.3, 4.5, 1.2, 6.8, 0.8, 4.9 };
bsort(array,n);   // sort the array
```

Const Modifiers and Pointers

- The use of the const modifier with pointers is confusing as it can mean two things
  - const int* cptrInt: // cptrInt is a pointer to a const int
    You can not change the value of the integer that cptrInt points to but you can change the pointer itself
  - int* const ptrcInt: // ptrcInt is a constant pointer to int
    You can change the value of the integer that ptrcInt points to but you can not change the pointer itself

Memory Management

- In order to create an array in C/C++ you have to know its size in advance during compile time, in other words it has to be a constant
- int size;
cout << “Enter size of array : “;
cin >> size;
int array[size];   // ERROR size has to be a constant
- Solution in C++, use vector class from the STL which is expandable
Memory Management

Date* void CreateDate() // allows the user to create a date object
{
    int day, month, year;
    char dummy;
    cout << "Enter dd/mm/yyyy: ";
    cin >> day >> dummy >> month >> dummy >> year;
    Date date(day, month, year);
    return &date;    // ERROR! Scope of date ends with end of function
}

Date *ptr;
ptr=CreateDate(); // call CreateDate() to generate a new date
cout << "You entered " << * ptr << endl;
// variable to which ptr points no longer exist, segmentation fault !!!

Memory Management

The new operator in C++ can be used to create objects that can be used after returning from a function.
Objects allocated in dynamic memory are called heap objects or to be "on free store" and have a permanent existence.

Date* CreateDate() // allows the user to create a date object
{
    int day, month, year;
    char dummy;
    cout << "Enter dd/mm/yyyy: ";
    cin >> day >> dummy >> month >> dummy >> year;
    return &date;    // ERROR! Scope of date ends with end of function
}

Date *ptr;
ptr=CreateDate(); // call CreateDate() to generate a new date
cout << "You entered " << * ptr << endl;   // ok, ptr refers to heap object

Memory Management

New can also be used to allocate blocks of memory.
The delete operator is used to release the memory allocated with new once it is no longer needed.

#include <cstring>
char *str = "This is an old C-style string";
int len=strlen(str); // computes the length of str
char *str
ptr = new char[len+1]; // set aside memory string + \0
strcpy(ptr,str); // copy str to new memory
cout << "ptr= " << ptr << endl;
delete [] ptr; // release ptr’s memory

New Operator in Constructors

class String // user-defined string class
{
    private:
        char* str; // pointer to block of characters
    public:
        String(char* s) // one-argument constructor
        {
            int length=strlen(s); // length of string argument
            str = new char[length+1]; // allocate memory
            strcpy(str,s); // copy argument to it
        }
        ~String() // destructor
        {
            delete [] str;
        }
        void Display()
        {
            cout << str << endl;
        }
};

String mystring="This is my string of Type String";
mystring.Display();
Pointers to Objects

- Pointers can point to objects as well as to built-in data types

```cpp
Date date;   // define a named Date object
date.Set(12,3,1996); // set the date
date.Display();  // display the date

Date *dateptr; // define a pointer to a Date object
dateptr=new Date; // points to new Date object
dateptr->Set(9,12,1999);  // set date using -> operator
dateptr->Display();  // display date
(*dateptr).Display();  // works as well but less elegant
```

Linked List Example

```cpp
struct link   // one element of list
{   // data item
    int data;
    link *next;  // pointer to next element
};

class linklist
{   // pointer to first link
    private:
        link* first;
    public:
        linklist() { first = NULL;}    // no argument constructor
        void additem(int d);        // add data item (one link)
        void display();          // display all links
};

void linklist ::additem(int d)   // add data item
{   // create a new link
    link* newlink = new link;   // give it data
    newlink->data = d;          // point to next link
    newlink->next=first;       // now first points to this link
    first = newlink;
}

void linklist ::display()    // display all links
{   // set ptr to first link
    link* current=first;    // until ptr points beyond last link
    while(current != NULL)   // print data
    {   // move to next link
        cout << current->data << " ";
        current=current->next;
    }
}
```

Linked List Example

```cpp
template <class T>
struct link   // one element of list
{   // data item
    T data;
    link *next;  // pointer to next element
};

template <class T>
class linklist
{   // pointer to first link
    private:
        link* first;   // no argument constructor
    public:
        linklist() { first = NULL;}    // add data item (one link)
        void display();          // display all links
};
```
**Linked List Example**

```cpp
template<class T>
void linklist<T>::additem(T t) // add data item
{
    link* newlink = new link; // create a new link
    newlink->data = t;          // give it data d
    newlink->next=first;       // it points to the next link
    first = newlink;               // now first points to this link
}
```

```cpp
template<class T>
void linklist<T>::display() // display all links
{
    link* current=first;    // set ptr to first link
    while(current != NULL)   // until ptr points beyond last link
    {
        cout << current->data << " ";   // print data
        current=current->next;            // move to next link
    }
}
```

**Self-Containing Classes**

A class might contain a pointer to an object of its own class but never an object of its own class

```cpp
class someclass
{
    someclass *ptr; // this is ok
};
class someclass
{
    someclass obj;  // ERROR! Can not do !
};
```

---

**Makefile**

A Makefile is a recipe for how to "cook" a product

The necessary operations are divided into single steps which partially depend on each other

Example: Change a flat tire on a car

Actions:
get_jack, get_spare_tire, lift_car, remove_flat_tire, attach_spare_tire, lower_car, stow_away_jack, stow_away_flat_tire, drive_away

Dependencies:
lift_car : get_jack
remove_flat_tire : lift_car
attach_spare_tire : get_spare_tire
lower_car : attach_spare_tire
stow_away_jack : lower_car
stow_away_flat_tire : remove_flat_tire
```

```bash
# include "mat.hh"
g++ -c lab1.cc
g++ -c mat.cc
g++ lab1.o mat.o -lm
```
In lab1 you have two source files mat.cc and lab1.cc from which you are supposed to create a program a.out

all: a.out

lab1.o: lab1.cc mat.h
g++ < lab1.cc

mat.o: mat.cc mat.h
g++ < mat.cc

a.out: lab1.o mat.o
g++ mat.o lab1.o -lm