CINCIS ITD

BASIC COMPONENTS
C++ PROGRAMS

PART 1

Beginners Guide

- See other presentation for quick review of C++ basics…

Assignment Statement

```
variable = expression;
```

- Expression is evaluated and its value is assigned to the variable on the left side: eg.
- num01 = 3
- second = ‘02’
- Str = ‘writing-a-string’
- double= 12.6;
- float = 17.4;

Preprocessor Directives - they begin with

- Many functions and symbols needed to run a C++ program - they are provided as collection of libraries
- Every library has a name and is referred to by a <u>*header file*</u>
- Preprocessor directives are commands given to the <u>C++ preprocessor</u>
- No semicolon at the end of these commands

Header files:

```
#include <headerFileName>
```

- Example:

```
#include <iostream>
```

  - Instructs the preprocessor to include the header file `iostream` in the program for `cin` and `cout`

Creating a C++ Program

- C++ program has two parts: (1) Preprocessor directives
- (2)the program
- Preprocessor directives and program statements make up the source code (.cpp)

Then ...

- Compiler generates the object code (.obj)
- Executable code is produced and saved in a file with extension .exe

Creating a C++ Program (continued)

- A C++ program is a made up of a collection of functions
- The function `main`
- The first line of the function `main` is called the heading of the function:

```
int main()
```

Structure of a program

- Typically, the first program beginners write is a program called "Hello World", which simply prints "Hello World" to your computer screen.
- Although it is very simple, it contains all the fundamental components C++ programs have:

Code:

```
first program in C++
#include <iostream>

int main()
{
  std::cout << "Hello World!";
}
```

Output:

```
Hello World!
```

Let's examine this program line by line:

Line 1: `// my first program in C++`

Two slash signs indicate line is a comment, in this case, it is a brief introductory description of the program.

Line 2: `#include <iostream>`

Lines beginning with a hash sign (`#`) are directives read and interpreted by the *preprocessor*. In this case, the directive `#include <iostream>`

Line 4: `int main ()`

This line initiates the declaration of a function main () of a type (`int`), a name (`main`) and a pair of parentheses (), optionally including parameters. The execution of all C++ programs begins with the `main` function.

Lines 5 and 7: `{` and `}`

The open brace (`{`) at line 5 indicates the beginning of `main`'s function definition, and the closing brace (`}`) at line 7, indicates its end.

Line 6: `cout << "Hello World!";`

This line is a C++ statement. Statements are executed in the same order that they appear within a function's body.

The most typical way to introduce visibility of these components is by means of *using declarations*:

```
using namespace std;
```

The above declaration allows all elements in the `std` namespace to be accessed in an *unqualified* manner (that is without the `std::` prefix).

so the last example can be rewritten to make unqualified uses of `cout` as:

```
// my second program in C++
#include <iostream>
using namespace std;

int main ()
{
  cout << "Hello World! ";
  cout << "I'm a C++ program";
}
```

```
Hello World! I'm a C++ program
```

Edit & Run

Use of Semicolons, Brackets, and Commas

- All C++ statements end with a semicolon
  - \- a statement terminator
- Brackets { and } are used , they are not C++ statements
- Commas are used to separate items in a list

Form and Style

- Consider two ways of declaring variables:
  - Method 1

```
int meter, cm;
double x, y;
```

  - Method 2

```
int a,b;double x,y;
```

- Both are correct; however, the second is hard to read

FUNCTIONS

PART 2

Summary

- Declarations
  - Definitions
  - Headers and the preprocessor
- Scope
- Functions
  - Declarations and definitions
  - Call by value, reference

Namespaces

Declarations – specify the type of a variable

- A declaration can also include the initializer value
- A name of a variable must be declared before it can be used ---
- Examples:
  - **int b = 10;** *// an int variable named ‘b’ is declared*
  - **const double cd = 9.7;** *// a double-precision floating-point constant*
  - **double sqr(double);** *// a function taking a double argument and // returning a double result*

Declarations and using header files

- Declarations are frequently introduced into a program through header files or “headers”
- --- providing an ‘interface’ to other parts of a program
- This allows for abstraction – so you don’t need to know the details of a function like **cout** in order to use it, for example when you add:

```
#include <iosteam>
```

to your code, the declarations in the file **std_lib_facilities.h** become available (including **cout**, etc.).

A Definition is when the declaration includes the full specification, or a value for the variable

- Examples of definitions

```
int a = 10;
int b;                  // an (uninitialized) int
vector<double> v;       // an empty vector of doubles
double sqr(double) { … };  // a function with a body
struct Pnt { int x; int y; };
```

- Examples of declarations that are not definitions

```
double sqrt(double);// function body missing
struct Point;            // class members specified elsewhere
```

Why both declarations and definitions?

- To refer to something, we need (only) its declaration
- In larger programs
  - Place all declarations in header files to ease sharing

- Often we want the definition “elsewhere”
  - Later in a file
  - In another file
- Declarations are used to specify the interfaces to codes
  - And to the libraries – so we can use code written by others

Beginner tutorial done

POINTERS

PART 3

Pointers

- POINTERs - How to Access Variables and Memory
- When declaring the variable, the computer associates the variable name with the particular location in memory
- and then stores a value at that location

Pointers

- So when you refer to the variable by name in the code, it takes two steps:

1. Look up the address that the variable name corresponds to
2. And goes to that location in memory to find or set the value it contains

- Pointers in C++ allows us to perform either one of these steps like a ‘handle’ or a way of accessing a variable with the & and * operators:

  - &x evaluates to the address of x in memory.

  - *(&x) takes the address of x and ‘**dereferences’ it** – this mean it retrieves the value at that location in memory.

………so   *(&x)   evaluates to the same thing as x.

Motivation: ’Memory addresses’ or Pointers

Memory addresses, or pointers, allow us to access and manipulate data in a much more flexible way –

- Manipulating the *memory addresses of data* can be done more effectively than manipulating the data itself.

Access Pointer Data by Using Deference Operator *

- Int main()
- int Age = 30;
- int DogAge = 9;

- int *pInteger = &Age;
- cout << “pInteger points to Age “;
- cout << *pInteger = << *pInteger << endl;
- pInteger = &DogAge;
- cout << pInteger << *pInteger << endl;
- return 0;
- }
- OUTPUT: pInteger points to Age
- pInteger = 0x0025F778
- *pInteger = 9

So what are Pointers?

- **Pointers are just variables that store integers**

–**these integers happen to be memory addresses,**

(and usually they are the addresses of other variables.)

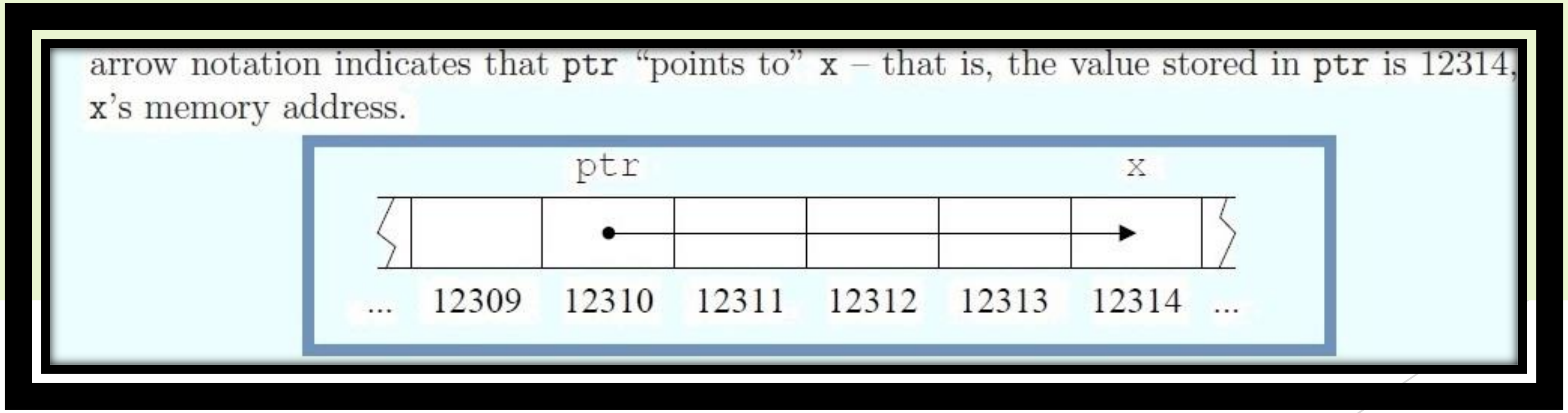
- Thus - a pointer that stores the address of variable x is said to point to ‘x.’

- Then we can access the value of ‘x’ by **dereferencing** the pointer.

Pointers and their Behavior

Pointers are similar in idea to arrays, with a row of adjacent cell locations in memory – see figure below:

With each cell representing one block of memory the pointer arrow notation in the figure shows how the pointer operates.


arrow notation indicates that **ptr** "points to" **x** – that is, the value stored in **ptr** is 12314, **x**'s memory address.

Pointer Syntax/Usage

Declaring Pointers:

Declare a pointer variable named *ptr* that points to an integer variable named *x*:

int * ptr = &x;

int *ptr declares the pointer to an integer value,

that is initializing to the address of *x*.

pointers are to values of any *type*, and declared as pointers to:

*data_type * pointer_name ;*

pointer name becomes the variable of the type

*data type **

- that is the pointer to a data type of the value - whatever…." "

Using Pointer Values

- Once a pointer is declared, it can be dereferenced with the * operator to access its value:

```
cout <<  * ptr;  //   value pointed to by ptr ,
                 // would be x's value
```

We can use deferenced pointers as values:

```
* ptr = 5;  //Sets the value of x
```

Pointer - Without the * operator

Without the * operator, the identifier x refers to the pointer itself, not the value it points to:

- cout << ptr; // Outputs memory address of x in base 16

Examples of some advantages:

- Easier to ‘pass-by-reference’ to functions
- Manipulate complex data structures, even if their data is scattered in different memory locations
- Use *polymorphism* - calling functions on data without knowing exactly what kind of data it is (we will see this later in slides)

Pointers as arguments

We can pass pointers as arguments to functions, just like any other data type,

The same way :

```
void func(int x) {...},
```

we can say :

```
void func(int *x){...}.
```

Pass-by-reference

Here is an example of using pointers to square a number in a similar way to a pass-by-reference function:

```
1 void squareByPtr ( int * numPtr ) {
2   * numPtr = * numPtr * * numPtr ;
3 }
4
5 int main () {
6   int x = 5;
7   squareByPtr (& x);
8   cout << x; // Prints 25
9 }
```

Note the varied uses of the * operator on line 2.

const Pointer

There are three places the `const` keyword can be placed within a pointer declaration...
=one for the pointer itself or one for the value it points to.

```
const int * ptr;
```

First : the above declares a changeable pointer to a *constant* integer and then cannot be changed through this pointer, while the pointer may be changed to point to another constant integer.

```
int * const ptr;
```

Second : declares a constant pointer and the integer value can be changed through this pointer, but the pointer may not be changed to point to another integer.

Third Case:

```
const int * const ptr;
```

forbids changing both the address and the value the pointer points to.

Null, Uninitialized, and Deallocated Pointers

- It there are pointers that do not point to any valid data this will mean if we are dereferencing such a pointer that it will create a runtime error.

- A pointer that is set to 0 is called a null pointer, this is an invalid pointer as there is no memory location ‘0’

References

The reference in the declaration:

```
void f(int &x){...}
```

and call `f(y)`,

the reference variable `x` becomes a label or 'tag' for the value of `y` in memory.

We can also declare a reference variable locally, as follows:

```
int  y;
int  &x  =  y;  //  Makes  x  a reference  to y
```

this means that changing `x` will change `y` and vice versa, because they are two names for the same thing.

So -references are just pointers that are dereferenced every time they are used.

The only differences between using pointers and using references are:

- You cannot change the location to which a reference points, but you can change the location to which a pointer points. Hence references must always be initialized when they are declared.
- When writing the value that you want to make a reference to, you do not put an & before it to take its address, but you do need to do this for pointers.

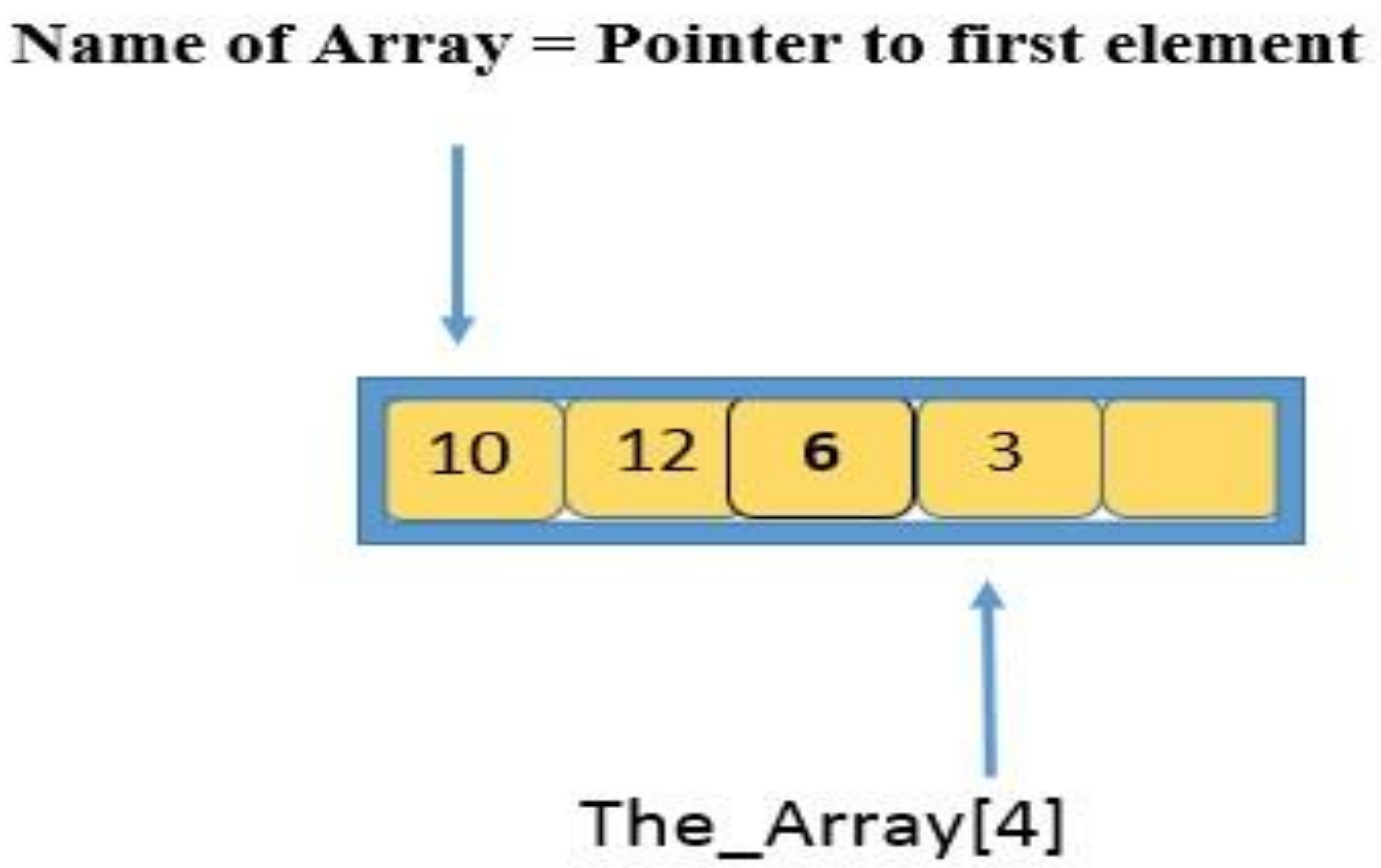
Use of the * operator

- 1. Declaration of the pointer : the * is placed before the variable name to indicate the variable name and type is a pointer eg. * int
- 2. pointer is set to some value * is put before the name to dereference it ie. To access or set values to it

Use of the & operator

- Indicate a reference data type
- int &x
- Take out the address of the variable
- int *ptr = &ptr;

Pointers and Arrays



More on Arrays…

- The array always starts at ‘0’ – not 1, as it is the element that is zero away from the start of the array

- The_Array[4] is four away from the start of the away
- Arrays are always passes by reference

Pointer Arithmetic

Pointer arithmetic is a way of using subtraction and addition of pointers to move around between locations in memory,

…..typically between array elements.

Adding an integer ‘*n*’ to a pointer produces a new pointer pointing to ‘*n*’ positions further down in memory.

Pointer Arithmetic

- Addition and subtraction can be used with pointers to move to new locations in memory
- adding an integer *n* to the pointer produces a new pointer pointing to the to new location at ‘n’

Add/subtract two pointers

- In line 3 of the code in previous slide ptr++ moves the pointer to the next element in the array, not just the next byte in memory, that is to the second element of the array

- Notice we can use ptr2 to find the number of array elements between ptr and ptr2 – to add and subtract operations

char * Strings

char * … A string is an array of characters

- When you set char * to a string it means you are setting a pointer to point to the first character in the array that holds the string

- To modify you modify the contents as an array of characters

Modify a string

```
char course_name01  =  { '3',  '5' ,'.' '9', '8','\0'}

char *course_name02 = "35.98";
```

we can modify the contents of course_name01 but get an error when attempting to modify the contents of course_name02

Using Declarations and Directives

- To avoid the tedium of
  - **std::cout << "Please enter stuff… \n";**

- ---- write a “using directive” for namespace

using namespace std;

cout << "Please enter stuff… \n"; *// accessing - std::cout*

cin >> x; ***// accessing --*** *std::cin*

CLASSES AND OOP

PART 4

The simplest Class (or a C-structure) can be thought of being a collection of variables of different types

```
structure Temperature {
    double degree;
    char scale;
};
```

A first simple ‘class’ or ‘object-oriented’ solution

```
class Temperature {
    public:
      double degree;
      char scale;
};
```

degree and scale are the two member variables

The dot operator for public members

- means that the member variables can be accessed from the objects

```
Temperature temp1, temp2;
temp1.degree=54.0;
temp1.scale='F';
temp2.degree=104.5;
temp2.scale='C';
```

Note - a C++ struct is a class in which all members are by default public.

Some basic operations:

```
void print(Temperature temp) {
        cout << "The temperature in degrees is "
                  << temp.degree <<  "with the selected scale is " <<
                   temp.scale << endl;
}
```

```
double celsius(Temperature temp) {
     double cel;
     if (temp.scale=='F') cel=(temp.degree-32.0)/1.8;
     else cel=temp.degree;
     return cel;
}
double fahrenheit(Temperature temp) {
   double fa;
   if(temp.scale=='C') fa= temp.degree *1.8+32.0;
   else fa=temp.degree;
   return fa;
}
```

An Example of the Application

```
Temperature year_temp[12];
```

```
double year_AverageCelsius(Temperature arraytemp[])
{
  double av=0.0;
   for (int i=0;i<12;i++)
       av=av+celsius(arraytemp[i]);
   return av;
};
```

Designing the CLASS - from variables and functions

Actual problem:

1. Member ‘variables’ are still separated from ‘functions’ manipulating these variables.

2. However, ‘functions’ are intrinsically related to the ‘type’. CHANGE THIS TEXT ……….

The simplest class defined this way is a collection of (member) variables that is very similar to structures from C

A more advanced class is a collection of (member) variables and (member) functions

“The art of programming is the art of organising complextity.”

Next - improving the Temperature class by associating the member functions

Assembly the data and operations together into a class!

```
class Temperature{
   public:
      void print();       // member functions
       double celsius();
       double fahrenheit();

      double degree;  // member variables
      char scale;
};
```

Operators for members

The dot operator not only for public member variables of an object, but also for public member functions (during usage), e.g.

```
Temperature temp1;
temp1.celsius();
temp1.print();
```

function → for a method

Function(procedure) call → for a message

Temp1 receives `print()` message and displays values stored in degree and scale variables

Operators for defining member functions

:: for member functions of a class - during definition

Functions full name

```
double Temperature::celsius() {
        double cels;
        If (scale==‘F’) cel= (degree-32.0)/1.8;
        else cels=degree;
        return cels;
}
```

:: is used with a class name - while dot is used with an object

Using the ‘private’ member modifier

‘private’ members can only be used by member functions,

This means the private member variables can be used for data protection and information hiding – and only member functions to access the private data instead

New version of Temperature class

```
class Temperature{
        public:   // member functions
            void print();
             double celsius();
             double fahrenheit();
        private:  // member variables
            double degree;
            char scale;
  };
```

When the datum ‘degree’ is private, it can not be accessed directly by using `temp1.degree`

```
double Temperature::celsius() {
         double cels;
         If (scale=='F') cels= (degree-32.0)/1.8;
         else cels=degree;
         return cels;
}
```

Possible only when ‘degree’ is public

OK

Private member variables can only be accessed by ‘member functions’ of the same class.

Using member functions to indirectly access private data

```
class Temperature{
        public:    // member functions
                    double get_Degree();
                char get_Scale();
             void set(double newDegree, char newScale);

                    void print();
                    double celsius();
                    double fahrenheit();
            private:   // member variables
            double degree;
            char scale;
    };
```

Some member functions on private data:

```
double Temperature::get_Degree() {
    return degree;
}
```

```
double Temperature::get_Scale() {
    return scale;
}
```

```
double Temperature::set_temp(double d, char s) {
  degree = d;
  scale = s;
}
```

Summary:

A collection of member variables and member functions is a Class

Struct is a class with only member variables, and all of them public

‘public’ member can be used outside by dot operator

But the ‘private’ members can only be used by member functions

```cpp
class A {
   public:
      void f();

      int x;
   private:
      int y;
}

void A::f() {
   x=10;
   y=100;
}
```

```cpp
int main() {
   A a;

   a.f();

   cout << a.x << endl;
   cout << a.y << endl; // cant do this

   a.x = 1000;
   a.y = 10000; // cant do this either
}
```

Some basic member functions:

Classification of member functions:

- ❑ Constructors = for the initialisation
- ❑ Access for - member variables
- ❑ Updating for modifying data
- ❑ I/O and utility functions …

A more complete definition of a class should have a complete set of basic member functions manipulating the class objects

```
class Temperature{
    public:
        Temperature();
        Temperature(double idegree, char iscale);
        double get_Degree() const;
        char get_Scale() const;
        void set(double newDegree, char newScale);
        void read();
        void print() const;
        double fahrenheit();
        double celsius();
    private:
        double degree;
        char scale;
};
```

The Constructor

A constructor has a name is always the same as the name of the class.

```
Temperature::Temperature(){
    degree = 0.0;
    scale = 'C';
}
```

A constructor function initializes the data members when a Temperature object is declared.

```
    Temperature temp3;
```

Constructor functions have no return type - not even void.

An Explicit-Value Constructor

```
Temperature::Temperature(double a, char b){
  degree = a;
  scale = toupper(b);
}
```

An explicit-value constructor initializes the data members – when a Temperature object is declared with the parameters:

```
Temperature temp3(98.6, 'F');
```

Constructor is ‘overloaded’, the same name but different arguments.

Data access (inspector) Functions

```
double Temperature::get_Degree() const {
   return degree;
}
char Temperature::get_Scale() const {
   return scale;
}
```

A data access function allows programmers to access to read (but not allowed to modify) data members of the class.

```
double a = temp1.getDegree();
char b = temp1.getScale();
```

Update Functions

```
void Temperature::set(double a, char b){
   degree = a;
   scale = toupper(b);
   if(scale!='C' && scale!='F'){
       cout << “Faulty Temperature scale: " << scale << endl;
       exit(1);
    }
}
```

The update function modifies data members of the class.

```
temp1.set(32, 'F');
```

Reading Temperature

```
void Temperature::read(){
 cin >> degree >> scale;
 scale = toupper(scale);
 if(scale!='C' && scale!='F'){
        cout << "Faulty Temperature scale: " << scale << endl;
        exit(1);
   }
}
```

Using the `read()` member function:

```
Temperature temp1;
cout << "Enter the temperature reading : (e.g., 98.6 F): ";
temp1.read();
```

When `temp1` receives the `read()` message input, it gets the values from `cin` into variables `degree` and `scale`.

Conversion functions - The member function `fahrenheit()` gives the degree and scale in Fahrenheit

```
double Temperature::fahrenheit(){
   double fahr;
   if(scale == 'C')
    fahr = degree*1.8+32.0;
   else fahr = degree;
   return fahr;
}
```

* **The `fahrenheit()` member function:**

```
Temperature temp1;    // default value: 0 C
cout << temp1.Fahrenheit();
```

* **When `temp1` receives the `fahrenheit()` message, it gets the Fahrenheit temperature 32 F.**

The member functions `celsius()`

```
void Temperature::celsius(){
      double cels;
      if(scale == 'F')cels = (degree- 32.0)/1.8;
      else cels = degree;

    return cels;
   }
```

Application of Temperature class

```
#include <iostream>
using namespace std;


// definition of Temperature class can go here  …

void main(){
   char resp;
   Temperature temp;
   do{
    cout << "Enter temperature (e.g., 98.6 F): ";
    temp.read();
    cout << temp.fahrenheit() << “Fahrenheit” << endl;
    cout << temp.celsius() << “Celsius” << endl;
    cout << endl << endl;
    cout << "Another temperature to convert? ";
    cin >> resp;
   }while(resp == 'y' || resp == 'Y');
}
```

The ‘Smart’ Temperature Object

- A smart object should carry within itself the ability to perform its operations
- Operations of `Temperature` object :
  - initialize degree and scale with default values
  - read a temperature from the environment and store it
  - compute the corresponding Fahrenheit temperature
  - compute the corresponding Celsius temperature
  - display the degrees and scale to the display

Object-Oriented Programming (OOP) and Inheritance

PART 5

OOP

- We have defined the composite datatypes using classes for C++
- Now we consider the programming philosophy object-oriented programming (OOP).

Approach of "procedural" programming languages

Classic "procedural" programming languages before C++ (such as C) approach was:

- Split it up into a set of tasks and subtasks
- Make functions for each of the tasks
- Instruct the computer to perform the tasks in sequence

Problem - the large amounts of data and tasks, makes for complex and programs that are very difficult to maintain.

Code for a program to model a company

- Consider the task of modeling the operation of a company.
- Such a program would have lots of separate variables storing information on various company departments, and there'd be no way to group together all the code that relates to, say, the staff.
- It's hard to keep all these variables and the connections between all the functions in synch.

Independent modular pieces of code

- To manage this complexity, it’s most effective to package up independent modular pieces of code.
- Then we think of the code in terms of interacting objects: we’d talk about interactions between the departments, the staff, the offices etc.
- OOP allows programmers to pack away details into neat, self-contained units (objects) and then that they can think of the object code more abstractly and focus on the interactions between them.

For OOP the primary ideas of it are:

- **Encapsulation:** the grouping of related data and functions together as objects and then defining an interface to those objects
- **Inheritance:** allowing code to be reused between related types
- **Polymorphism:** allowing a value to be one of several types, so that it is determined at runtime which functions call on it based on its type

Encapsulation

Encapsulation just refers to C++ packaging related -- grouping together.

- C++ classes represent how data is packaged up and the operations supported.
- This means if we look at a class, we do not need to know how it actually works to use it; all we need to know about is its public methods/data –that is its interface.

public and private access specifiers

- Similar to driving a car, the steering wheel is the interface to driving the car, you do not need to know all about the engine parts to drive around.

- **In C++ you specify public and private access specifiers**

- the things you define in a class are internal details which someone using your code should not have to worry about, and this practice of hiding away these details from client code is called "data hiding," or making your class a "black box."

what happens in an object-oriented program

- One way to think about what happens in an object-oriented program is :

- we define what objects exist and what each one knows,

- then the objects send messages to each other - by calling each other's methods to exchange information and tell each other what to do.

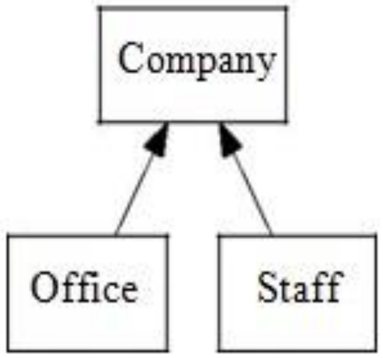
Class Office inherits from class Company.

- Now class Office has all the data members and methods of class Company, as well as a style data member and a getStyle method.
- This is equivalent to saying that **Office** is a **derived class**, while Company is its **base class**.
- You may also hear the terms **subclass** and **superclass** instead.

Derived Classes

Similarly the Staff class inherits from Company and shares its code in the same way as the Office class.

This would give a class hierarchy like the following:



Class hierarchies are generally drawn with arrows pointing from derived classes to base classes.

OO Instances

Is-a vs. Has-a

There are two ways we could describe some class A as depending on some other class B:

1. Every A object **has a** B object. For example every `Company` object has a `string` object (called `premises`).
2. Every instance of A **is a** B instance. For example every `Office` is a `Company`, as well.

BUT - "Has-a" relationships should be implemented by declaring data members, not by inheritance as it only allows us to define "is-a" relationships, but it should not be used to implement "has-a" relationships.

Overriding the Method

- We might want to generate the description for `Office` class in a different way from generic Company class
- To achieve we can simply redefine the `get_Desc` method in `Office`, as below.
- Then, when we call `get_Desc` on a Office object, it will use the redefined function.
- Redefining in this way is called 'overriding' the function.

Overriding functions

```
class Office : public Company { //Office inherit from Company
 string style;

 public:
   Office(const string &myPremises, const int myYear,
              const string &myStyle)
            : Company (myPremises, myYear), style(myStyle) {}
    const string get_Desc() // Overriding this member function
          { return stringcnv(year) + ' '+ style + ":" + premises
      ;}
            const string &getStyle() { return style;}
      };
```

Programming by Difference

- **The use of inheritance is in its defining of derived classes - we only need to specify what's different about them from their base classes.**
- **This is a powerful technique is called programming by difference. So we can *reuse code* from previous projects and extend it with this technique.**
- **Inheritance allows only overriding methods and adding new members and methods. We cannot remove functionality that was present in the base class – so our previous work can be *protected* on projects that are to be extended also.**

Access Modifiers and Inheritance

- If we'd declared year and premises as private in Company, we wouldn't be able to access them - even from a derived class like Office.
- Declaring them as **`protected`** - will allow derived classes only but not outside code to access data members and member functions
- The **`public`** keyword used in specifying a base class (e.g., class Office : public Company {...})
- which means that inherited methods declared as public are still public in the derived class. Specifying protected would make inherited methods, even those declared public, still have the protected visibility.

Polymorphism

- The word polymorphism means having many ‘forms’ and it refers to the ability of one object to have many types.
- Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.
- This means if we have a function that expects a `Company` object, we can safely pass it a `Office` object, because every `Office` is also a `Company`.
- And also for references and pointers: anywhere you can use a `Company *`, you can also use a `Office *`.

`virtual` Functions

- The following example – the *vPtr is call the Company version of itself even though the object it points to is actually an Office:

```
1. Office c ("CAPEL BUILDING", 2015);
2. Company *vPtr = &c;
3. cout -> get_Desc();
```

- the

`virtual` Functions (cont)

- Because vPtr is declared as a Company *, this will call the Company version of get_Desc function, - even though the object pointed to is actually a Office.
- Usually we'd want the program to select the correct function at runtime based on which kind of object is pointed to. We can get this behavior by adding the keyword **`virtual`** before the method definition:

`virtual` Functions (cont)

…adding the keyword `virtual` before the method definition,

With this definition, the code below will correctly select the Office version of get_Desc () function

```
class Company {
....
    virtual const string get_Desc() {...}
};
```

Dynamic dispatch – with references

- Because references are implicitly using pointers, the same issues apply to references

```
1. Office c ("CAPEL BUILDING", 2015);
2. Company &v = c;
3. cout << v. get_Desc();
```

Method is declared as `virtual`

- Once a method is declared `virtual` in some class **C**, it is virtual in every *derived* class of **C**, even if not explicitly declared as such.
- Thus - we will only call the Office version of get_Desc() function if get_Desc() is declared as `virtual` first.
- However, it is a good idea to declare it as virtual in the derived classes anyway for clarity.

Pure `virtual` functions

- Arguably we may not want a way to define get_Desc() for a ‘generic’ Company –
- We will only want the derived classes for a definition of it, since there is no such thing as a generic Company that doesn’t also have an Office, and Staff, etc.
- But at the same time we may not want to require every derived class of Company to have this function either.

get_Desc() - will only create derived classes

- So omit the definition of **`get_Desc()`** from **`Company`** by making the function pure virtual via the following syntax, the = 0 indicates that no definition will be given.

```
class Company  {
     ...
  virtual const string get_Desc()= 0;  // Pure  virtual
 };
```

Company is then an abstract class

- This implies that one can no longer create an instance of Company; one can only create instances of Offices, Staffs, and other derived classes which do implement the get_Desc method.
- Company has then become an **abstract** class – one which defines only *an interface*, but doesn't actually implement it,
- and therefore cannot be instantiated.

Multiple Inheritance

- C++ allows a class to have multiple base classes:
- This specifies that Office should have all the members of both the **Company** and the **NGO** classes.

```
class Office  : public Company , public NGO  {
      ...
};
```

Multiple inheritance is difficult and potentially dangerous

If both Company and NGO define a member x, you must remember to clarify which one you're referring to by saying **`Company::x`** or **`NGO::x`**.

-
  - If both **`Company`** and **`NGO`** inherited from the same base class, you'd end up with two instances of the base class within each Office (a "dreaded diamond" class hierarchy).
- In general, avoid multiple inheritance unless you are sure of exactly want to do with it.

STANDARD TEMPLATE LIBRARY - VECTORS

PART 6

VECTORS - Introduction to the STL

The *Standard Template Library* (or STL) is a collection of data types and algorithms that you to use in your programs.

VECTORS

- The data types that are defined in the STL are called *containers* - they store and organize data.
- There are two types of containers in the STL - they are the *sequence containers* and *associative containers*.
- The `vector` data type is a sequence container.

The STL `vector` is similar to the `array`…

- A `vector` holds a sequence of sequence of values, or elements :
- A `vector` stores its values or its elements in contiguous memory locations.
- We use the array subscript operator `[]` to read the individual elements in the `vector`

- However, a `vector` offers several advantages over arrays - **this is very important to note**:
- Here are a few:
  - **You do not have to declare the number of elements that the vector will have.**
  - **If you add a value to a vector that is already full, the vector will automatically increase its size to accommodate the new value.**
  - **`vectors` can report the number of elements they contain.**

Declaring a `vector`

- To use vectors in your program, you first `#include` the vector header file with :

```
#include <vector>
```

*<u>Notice: </u>There is no **.h** at the end of the file name.*

Declaring a `vector`

- The next step is to include after other your `#include` statements:

```
using namespace std;
```

Declaring a `vector`

```
vector<int> numbers;
```

The statement above declares `numbers` as a `vector` of `int`s.

To declare a starting size as follows.

```
vector<int> numbers(10);
```

The statement above declares `numbers` as a `vector` of 10 `int`s.

Other examples of vector Declarations

DECLARATION	Description of the declaration
`vector<float> things;`	Declares things as an empty vector of floats.
`vector<int> results(12);`	Declares results as a vector of 12 ints.
`vector<char> emails(20, 'A');`	Declares emails as a vector of 20 characters. Each element is initialized with 'A'.
`vector<double> ads_2(ads_1);`	Declares ads as a vector of doubles. All the elements of ads_1, which also a vector of doubles, are copied to ads_2.

Storing and Retrieving vals_in_vector in a `vector`

- To store a value in an element that already exists in a `vector`, you may use the array subscript operator `[]`.

Example - Program

```
// This program stores, in two vectors, the hours worked by 5
// employees, and their hourly pay rates.

#include <iostream>using namespace std;
#include <vector>  // Needed to declare vectors
using namespace std;

int main()
{
    vector<int> hourly(40);              // Declare a vector of 40 integers
    vector<float> pay_Rates(5);   // Declare a vector of 5 floats

    cout << "Enter the hours worked by 5 employees and their\n";
    cout << "hourly rates.\n";
    for (int index = 0; index < 5; index++)
    {
        cout << "Hours worked by employee #" << (index + 1);
        cout << ": ";
        cin >> hours[index];
        cout << "Hourly pay rate for employee #";
        cout << (index + 1) << ": ";
        cin >> pay_Rates[index];
    }
```

Example - Program *(cont)*

```
   cout << "Here is the gross pay for each
employee:\n";
   cout.precision(2);
   cout.setf(ios::fixed | ios::showpoint);
   for (index = 0; index < 5; index++)
   {
     float Total_Pay = hours[index]* pay_Rates[index];
      cout << "Employee #" << (index + 1);
      cout << ": $" << Total_Pay << endl;
   }
   return 0;

}
```

Example - Program

Program Output with Example Input Shown in Bold

```
Enter the hours worked by 5 employees and their
hourly rates.
Hours worked by employee #1: 10 [Enter]
Hourly pay rate for employee #1: 9.75 [Enter]
Hours worked by employee #2: 15 [Enter]
Hourly pay rate for employee #2: 8.62 [Enter]
Hours worked by employee #3: 20 [Enter]
Hourly pay rate for employee #3: 10.50 [Enter]
Hours worked by employee #4: 40 [Enter]
Hourly pay rate for employee #4: 18.75 [Enter]
Hours worked by employee #5: 40 [Enter]
Hourly pay rate for employee #5: 15.65 [Enter]
Here is the gross pay for each employee:
Employee #1: $97.50
Employee #2: $129.30
Employee #3: $210.00
Employee #4: $750.00
Employee #5: $626.00
```

Using the `push_back` Member Function

- You cannot use the `[]` operator to access a `vector` element that does not exist, so we can use the `push_back` member function to store a value in a `vector` that does not have a starting size, or is already full

```
// This program stores, in two vectors, the hours worked by a
specified
// number of employees, and their hourly pay rates.

#include <iostream>using namespace std;
#include <vector> // Needed to declare vectors
using namespace std;

int main()
{
   vector<int> hours;        // hours is an empty vector
   vector<float> pay_Rate;  // pay_Rate is an empty vector
   int num_of_Employees;        // number of employees

   cout << "How many employees do you have? ";
   cin >> num_of_Employees;
   cout << "Enter hours worked:" << num_of_Employees;
   cout << " employees and the hourly rates.\n";
```

```
    for (int index = 0; index < num_of_Employees; index++)
    {
        int tempHours;   // To hold the number of hours entered
        float tempRate; // To hold the payrate entered

        cout << "Hours worked by employee #" << (index + 1);
        cout << ": ";
        cin >> tempHours;
        hours.push_back(tempHours); // Add an element to hours
        cout << "Hourly pay rate for employee #";
        cout << (index + 1) << ": ";
        cin >> tempRate;
        payRate.push_back(tempRate); // Add an element to payRate
    }
    cout << "Here is the gross pay for each employee:\n";
    cout.precision(2);
    cout.setf(ios::fixed | ios::showpoint);
    for (index = 0; index < num_of_Employees; index++)
    {
            float Total_Pay = hours[index] * payRate[index];
        cout << "Employee #" << (index + 1);
        cout << ": $" << Total_Pay << endl;
    }

    return 0;

}
```

Determining the size of a `vector with .size()`

- Unlike arrays, vectors can report the number of elements they contain.

```
num_of_vals_in_vector = vector_1.size();
```

- The size is returned with the `size` member function. Here is an example of a statement that uses the size member function:

Example – code to show a vectors size

```
void show_vector_size(vector<int> vect)
{
  for (int count = 0; count < vect.size(); count++)
       cout << vect[count] << endl;
}
```

Using push_back() to push vals_in_vector on to the vector

```
// This program demonstrates the vector size
// member function.
#include <iostream>

using namespace std;

#include <vector>
using namespace std;


// Function prototype
void show_push_back(vector<int>);

int main()
{
    vector<int> vals_in_vector;

    for (int count = 0; count < 7; count++)
        vals_in_vector.push.back(count * 2);
    show_push_back (vals_in_vector);

    return 0;

}
```

push_back()

```
//*****************************************************
// Definition of function show_push_back_in_vector.*
// This function accepts an int vector as its      *
// argument. The value of each of the vector's     *
// elements is displayed.                          *
//*****************************************************

void show_push_back(vector<int> vect_01)
{
   for (int count = 0; count < vect_01.size(); count++)
      cout << vect_01[count] << endl;
}
```

Program Output

```
0
2
4
6
8
10
12
```

Removing Elements from a `vector`

- Use the `pop_back` member function to remove the last element from a `vector`.

```
retrieve_item.pop_back();
```

The statement above removes the last element from the `retrieve_item` vector.

push_back() and pop_back()

```
// This program demonstrates the vector size member function.

#include <iostream>

using namespace std;
#include <vector>
using namespace std;

int main()
{
    vector<int> vals_in_vector;

    // Store vals_in_vector in the vector
    vals_in_vector.push_back(1);
    vals_in_vector.push_back(2);
    vals_in_vector.push_back(3);
     cout << "The size of vals_in_vector is " << vals_in_vector.size()
<< endl;

    // Remove a value from the vector
    cout << "Popping a value from the vector...\n";
    vals_in_vector.pop_back();
    cout << "The size of vals_in_vector is now " <<
vals_in_vector.size() << endl;
```

pop_back()

```
    // Now remove another value from the vector
    cout << "Popping a value from the vector...\n";
    vals_in_vector.pop_back();
    cout << "The size of vals_in_vector is now " << vals_in_vector.size() << endl;

    // Remove the last value from the vector
    cout << "Popping a value from the vector...\n";
    vals_in_vector.pop_back();
    cout << "The size of vals_in_vector is now " << vals_in_vector.size() << endl;
    return 0;

}
```

Program Output

```
The size of vals_in_vector is 3
Popping a value from the vector...
The size of vals_in_vector is now 2
Popping a value from the vector...
The size of vals_in_vector is now 1
Popping a value from the vector...
The size of vals_in_vector is now 0
```

Clearing a `vector`

- To completely clear the contents of a `vector`, use the `clear` member function.

Eg. **`vector_02.clear();`**

- the `numbers` vector will be cleared of all its elements.

Example

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

int main()
{
   vector<int> vals_in_vector(100);

   cout << "The vals_in_vector vector has ‟
   << vals_in_vector.size() << " members of vector.\n";
   cout << "I will call the clear vector STL function...\n";
   vals_in_vector.clear();
   cout << "Now, the vals_in_vector vector has ‟
      << vals_in_vector.size() << " members.\n";
   return 0;

}
```

Program Output Display

```
The vals_in_vector vector has 100 elements.

I will call the clear vector STL function

Now, the vals_in_vector vector has 0 elements.
```

Detecting an Empty **`vector`** – `empty function – returns true/false if elements stored in vector`

- Here is an example of its use:

```
if (set.empty())
   cout << "No vals_in_vector in set.\n";
```

Program to show the empty function main ()

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

// Function prototype
float avg_Vector(vector<int>);

int main()
{
   vector<int> vals_in_vector;
   int numvals_in_vector;
   float average;

   cout << "How many vals_in_vector do you wish to
average? ";
   cin >> numvals_in_vector;
```

(cont)

```
for (int count = 0; count < numvals_in_vector; count++)
    {
        int tempValue;

        cout << "Enter a value:  ";
        cin >> tempValue;
        vals_in_vector.push_back(tempValue);
    }
    average = avgVector(vals_in_vector);
    cout << "Average:  " << average << endl;

    return 0;
}
```

```
//**********************************************************************
// Definition of function avg_Vector.                                  *
// This function accepts an int vector as its argument. If             *
// the vector contains vals_in_vector, the function returns the        *
// average of those vals_in_vector. Otherwise, an error message is     *
// displayed and the function returns 0.0.                             *
//**********************************************************************
```

```
float avg_Vector(vector<int> vect)
{
   int total = 0;   // accumulator
   float avg;    // average

   if (vect.empty())  // Determine if the vector is empty
   {
      cout << "No vals_in_vector to average.\n";
      avg = 0.0;
   }
   else
   {
      for (int count = 0; count < vect.size(); count++)
         total += vect[count];
      avg = total / vect.size();
   }
   return avg;
}
```

Program Output

```
How many vals_in_vector do you wish to average?
Enter a value: 12
Enter a value: 18
Enter a value: 3
Enter a value: 7
Enter a value: 9
Average: 9


How many vals_in_vector do you wish to average?
0
No vals_in_vector to average.
Average: 0
```

Summary of vector member functions

Member Function	Description
`at(`*`element`*`)`	Returns the value of the element located at *element* in the vector. `x = vector_01.at(6);` The above assigns the value of the 6th element of `vector_01` to `x`.
`capacity()`	Returns the maximum number of elements that may be stored in the vector without additional memory being allocated. `x = vector_01.capacity();` The statement above assigns the capacity of `vector_1` to `x`. .- note - this is not the same value as returned by the size member function

Vector member fns - clear(), empty(), pop_back()

`clear()`	Clears a vector of all its elements. `Vector_01.clear();` The statement above removes all the elements from `Vector_01`.

`empty()`	Returns true if the vector is empty. Otherwise, it returns false.
	`if (Vector_01.empty())`
	`     cout << "The vector is empty.";`
	The statement above displays the message if `Vector_01` is empty.

`pop_back()`	Removes the last element from the vector. `Vector_01.pop_back();` The statement above removes the last element of `Vector_01`, thus reducing its size by 1.

Summary of vector member functions

`push_back(`*`value`*`)`	Stores a value in the last element of the vector, unless it is full or empty, then a new element is created. `vect.push_back(9);` The statement above stores 9 in the last element of `vect`.
`reverse()`	Reverses the order of the elements in the vector – so the last element becomes the first element, and the first element becomes the last element, `vect.reverse();` The above reverses the order of the elements in `vect`.
`resize(`*`elements`*`, `*`value`*`)`	Resizes a vector by *elements* elements. Each of the new elements is initialized with the value in *value*. `vect.resize(9, 1);` The statement above increases the size of `vect` by 9 elements. The 9 new elements are initialized to the value 1.

vector function – swap ()

`swap(vector2)`	Swaps the contents of the vector1 with the contents of *vector2*. `Vect_01.swap(vect_02);` The statement above swaps the contents of `vect_01` and `vect02`.

VECTORS AND THE FREE STORE

PART 7

Overview

- Vector - How are they implemented?
- Pointers and free store
  - Allocation (new)
  - Access
    - Arrays and subscripting: []
    - Dereferencing: *
  - Deallocation (delete)

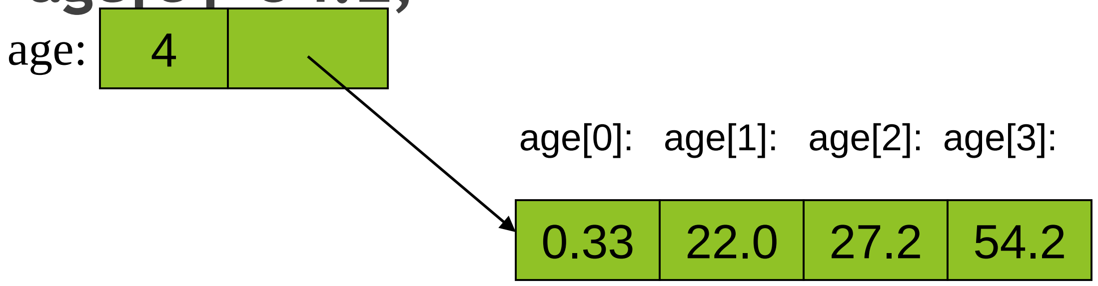
Vectors

- Vector is the most useful container
  - Simple
  - Compactly stores elements of a given type
  - With efficient access
  - Expands to hold any number of elements

Vector -Can hold an arbitrary number of elements

- A **vector**
  - Up to whatever physical memory and the OS can handle
- That number can vary over time
  - E.g. by using **push_back()**
- Example:

```
vector<double> age(4);
age[0]=.33;    age[1]=22.0;    age[2]=27.2;
  age[3]=54.2;
```



Vector

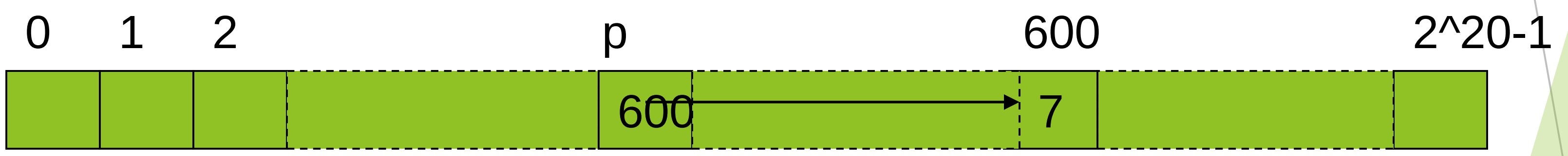
```
// a very simplified vector of doubles (like vector<double>):
class vector {
    int siz;          // the number of elements ("the size")
    double* elemt;    // pointer to the first elemtent
public:
    vector(int s);    // constructor: allocate s elements,
                      // let elemt point to them,
                      // store s in siz
    int size() const { return siz; }// the current size
};
```

- * means "pointer to" -- therefore: **double*** is a "pointer to **double**"
  - How do we make a pointer "point to" elements?
  - How do we "allocate" elements?

- Pointer values are memory addresses --they are a kind of integer value..

The first byte of memory is 0, the next 1, and so on

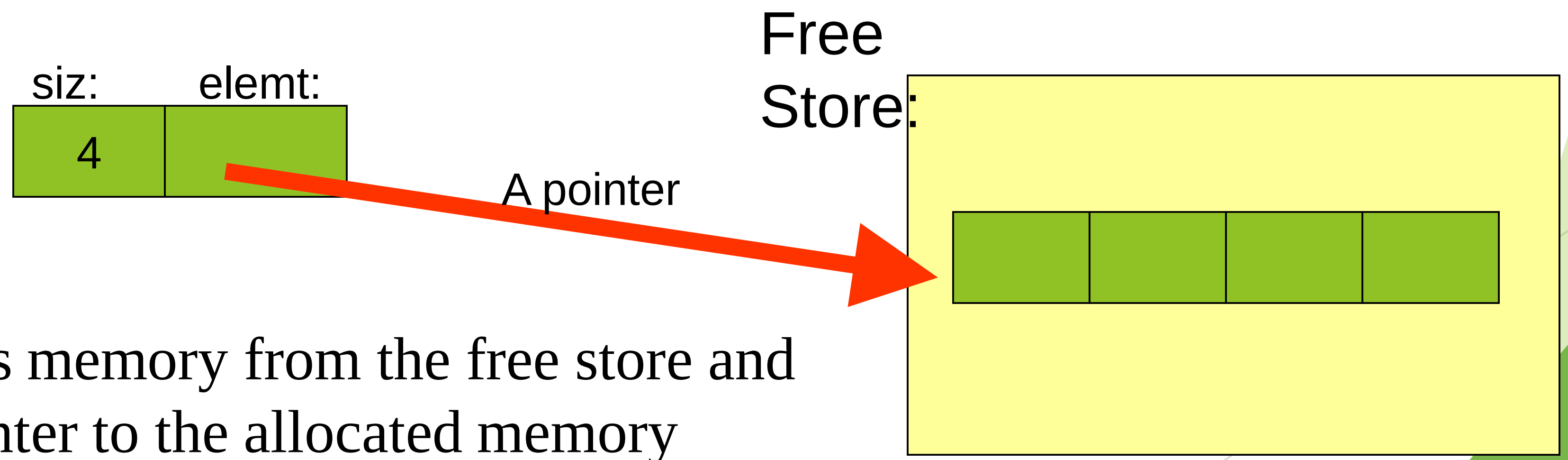
-- A pointer **p** can hold the address of a memory location



- A pointer points to an object of a given type
  - E.g. a **double*** points to a **double**, not to a **string**

Vector (constructor)

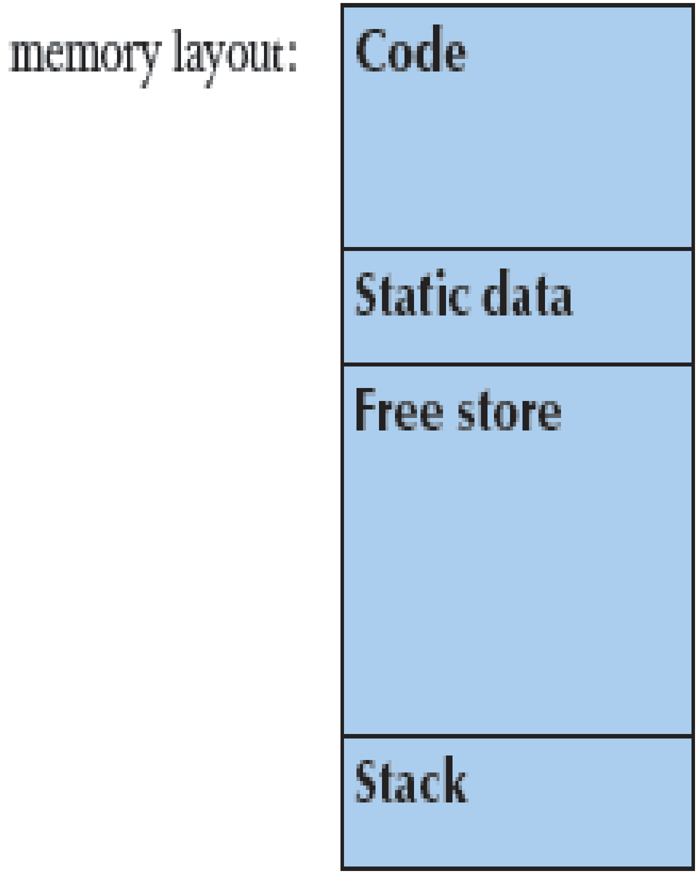
```
vector::vector(int s)  // vector's constructor
     :siz(s),           // store the size s in siz
      elemt(new double[s])// allocate s doubles on the free store
                          // store a pointer to those doubles in elemt
{
}
// Note: new does not initialize elements (but the standard vector does)
```



new allocates memory from the free store and returns a pointer to the allocated memory

The computer's memory

- As a program sees it:
  - Local variables "live on the stack"
  - Global variables are "static data"
  - The executable code is in "the code section"



The free store (sometimes called "the heap")

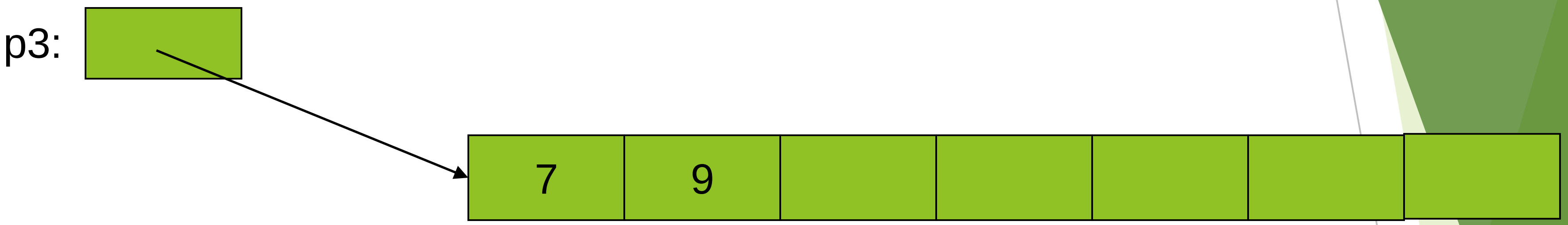
- You request memory "to be allocated" "on the free store" by the **new** operator
  - The **new** operator returns a pointer to the memory
  - A pointer is the address of the first byte of the allocated memory
  - **int* p = new int;**   *// allocate one uninitialized* ***int***
                  // ***int**** *means "pointer to* ***int****"*
    - **int* q = new int[8];**   *// allocate seven uninitialized* ***ints***
                    // "an array of 8 ***ints****"*
    - **double* pd = new double[n];** *// allocate* ***n*** *uninitialized* ***doubles***
  - A pointer points to an object of its specified type
  - But - the pointer does ***not*** know how many elements it points to

Access

- Individual elements

```
int* p1 = new int;        // get (allocate) a new uninitialized int
int* p2 = new int(5);        // get a new int initialized to 5

int x = *p2;          // get/read the value pointed to by p2
                      // (or "get the contents of what p2 points to")
                      // in this case, the integer 5
int y = *p1;          // undefined: y gets an undefined value; don't do that
```

Access



- Arrays (as a sequences of elements)

  int* p3 = new int[5]; *// get (allocate) 5* ***ints***

  // array elements are numbered [0], [1], [2], …

  p3[0] = 7; *// write to ("set") the 1st element of p3*

  p3[1] = 9;

  int x2 = p3[1]; *// get the value of the 2nd element of p3*

  int x3 = *p3; *// we can also use the dereference operator * for an array*

  *// *p3 means p3[0] (and vice versa)*

Why use free store?

- *To allocate objects that have to outlive the function that creates them*

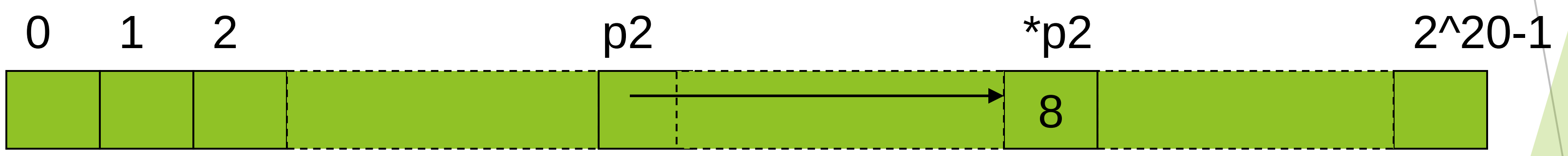
- For example

```
double* make(int n)  // allocate n ints
{
    return new double[n];
}
```

  - Another example: vector's constructor

Pointer values – are just memory addresses

```
// you can see a pointer value (but you rarely need/want to):
int* p1 = new int(8);           // allocate an int and initialize it to 8
double* p2 = new double(8);     // allocate a double and initialize it to 8.0
```

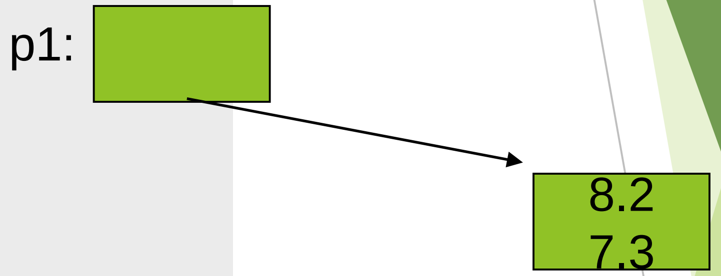


The output is as follows:

```
cout << "p1==" << p1 << " *p1==" << *p1 << "\n";    // p1==??? *p1==c
cout << "p2==" << p2 << " *p2==" << *p2 << "\n";    // p2==??? *p2=7
```

Access -- The pointer does not know the number of elements that it's pointing to (only the address of the first element)

```
double* p1 = new double;
*p1 = 6.3;        // ok
p1[0] = 8.2;          // ok
p1[17] = 6.4;         // ouch! Undetected error
p1[-4] = 2.4;         // ouch! Another undetected error
```




```
double* p2 = new double[100];
*p2 = 6.3;        // ok
p2[22] = 19.4;        // ok
p2[-6] = 2.6;         // ouch! Undetected error
```

Pointers, arrays, and vector

- Quote from Bjarne Stroustup:
- “
  - ***With pointers and arrays we are "touching" hardware directly with only the most minimal help from the language.***
  - ***Here is where serious programming errors can most easily be made, resulting in malfunctioning programs and obscure bugs***
    - *Be careful and operate at this level only when you really need to*
    - *If you get "segmentation fault", "bus error", or "core dumped", suspect an uninitialized or otherwise invalid pointer*
  - ***vector is one way of getting almost all of the flexibility and performance of arrays with greater support from the language (read: fewer bugs and less debug time). “***

Vector (construction and primitive access)

```
// a very simplified vector of doubles:
class vector {
    int siz;            // the size
    double* elemt;      // a pointer to the elements
public:
    vector(int s) :siz(s), elemt(new double[s]) { }  // constructor
    double get(int n) const { return elemt[n]; }     // access: read
    void set(int n, double v) { elemt[n]=v; }        // access: write
    int size() const { return siz; }                 // the current size
};

vector v(10);
for (int i=0; i<v.size(); ++i) { v.set(i,i); cout << v.get(i) << ' '; }
```

A problem: memory leak

```
double* calculate(int result_size, int max)
{
    double* p = new double[max];      // allocate another max doubles
                                      // i.e., get max doubles from the free store
    double* result = new double[result_size];
    // … use p to calculate results to be put in result …
    return result;
}


double* r = calculate(200,100);// oops! We "forgot" to give the memory
                               // allocated for p back to the free store
```

- Lack of de-allocation (usually called "memory leaks") can be a serious problem in real-world programs
- A program that must run for a long time can't afford any memory leaks

A problem: memory leak

```
double* calculate(int result_size, int max)
{
    int* p = new double[max];     // allocate another max doubles
                                  // i.e., get max doubles from the free store
    double* result = new double[result_size];
    // … use p to calculate results to be put in result …
    delete[ ] p;                  // de-allocate (free) that array
                                  // i.e., give the array back to the free store
    return result;
}

double* r = calculate(200,100);
// use r
delete[ ] r;            // easy to forget
```

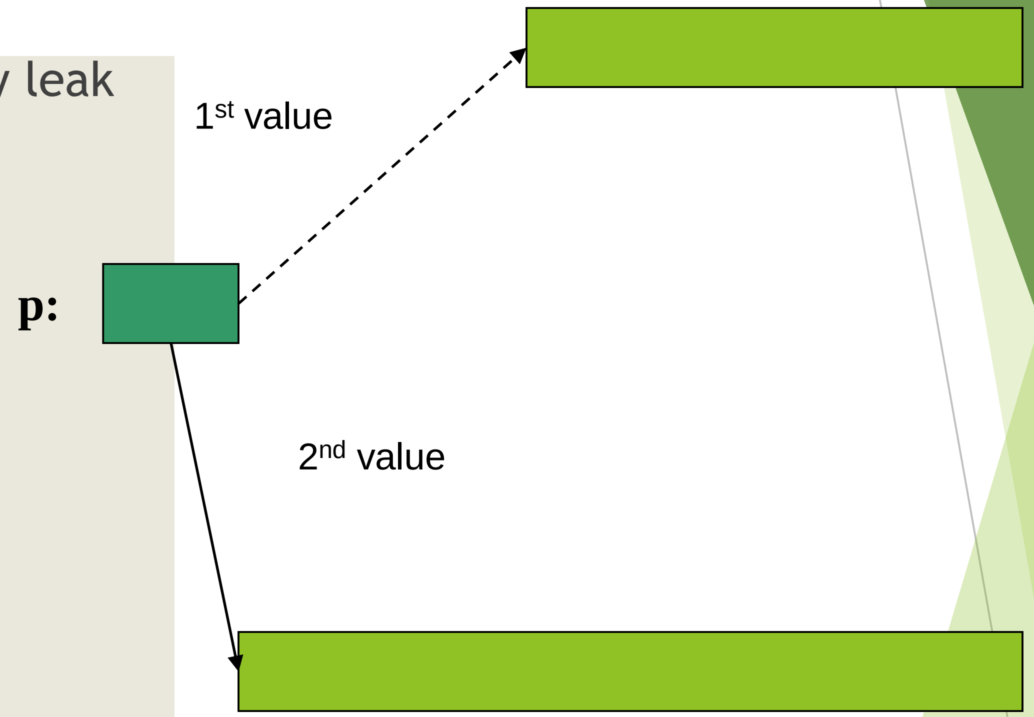
Memory leaks - if program runs forever then cant afford memory leaks

  - Eg. - An operating system is an example of a program that "runs forever"
- If a function leaks 8 bytes every time it is called, how many days can it run before it has leaked/lost a megabyte?
  - Answer – approx. - 130,000 calls
- But -- All memory is returned to the system at the end of the program If you run using an operating system (Windows, Unix, whatever), so the program that runs to completion with predictable memory usage may leak without causing problems
  - *i.e.*, memory leaks aren't "good/bad" but they can be a major problem in specific circumstances

Memory leaks

- Another way to get a memory leak

```
void f()
{
    double* p = new double[27];
    // …
    p = new double[42];
    // …
    delete[] p;
}

// 1st array (of 27 doubles) leaked
```



Memory leaks –how to avoid?

- don't mess directly with **new** and **delete**
  - Use **vector**, etc.
- Or use a garbage collector
  - A garbage collector is a program the keeps track of all of your allocations and returns unused free-store allocated memory to the free store
  - Unfortunately, even a garbage collector doesn’t prevent all leak (not covered here)

A Memory Leak Problem

```
void f(int x)
{
    vector v(x); // define a vector
            // (which allocates x doubles on the free store)
    // ... use v ...

    // give the memory allocated by v back to the free store
    // but how? (vector's elemt data member is private)
}
```

Vector (destructor)

```
// a very simplified vector of doubles:
class vector {
    int siz;            // the size
    double* elemt;      // a pointer to the elements
public:
    vector(int s)       // constructor: allocates/acquires memory
      :siz(s), elemt(new double[s]) { }
    ~vector()           // destructor: de-allocates/releases memory
      { delete[ ] elemt; }
    // …
};
```

- **Note: this is an example of a general and important technique:**
  - acquire resources in a constructor
  - release them in the destructor
- **Examples of resources: memory, files, locks, threads, sockets**

A problem: memory leak

```
void f(int x)
{
    int* p = new int[x];      // allocate x ints
    vector v(x);              // define a vector (which allocates another x ints)
    // … use p and v …
    delete[ ] p;    // deallocate the array pointed to by p
    // the memory allocated by v is implicitly deleted here by vector's destructor
}
```

- The **delete** now looks verbose and ugly
  - How do we avoid forgetting to **delete[ ] p**?
  - Experience shows that we often forget
- Prefer **deletes** in destructors

Free store summary

- Allocate using **new**
  - New allocates an object on the free store, sometimes initializes it, and returns a pointer to it
    - **int* pi = new int;** *// default initialization (none for* ***int****)*
    - **char* pc = new char('a');** *// explicit initialization*
    - **double* pd = new double[10];** *// allocation of (uninitialized) array*
  - New throws a **bad_alloc** exception if it can't allocate (out of memory)
- Deallocate using **delete** and **delete[ ]**
  - **delete** and **delete[ ]** return the memory of an object allocated by **new** to the free store so that the free store can use it for new allocations
    - **delete pi;** *// deallocate an individual object*
    - **delete pc;** *// deallocate an individual object*
    - **delete[ ] pd;** *// deallocate an array*
  - Delete of a zero-valued pointer ("the null pointer") does nothing
    - **char* p = 0;** *// C++11 would say* **char* p = nullptr;**
    - **delete p;** *// harmless*

void* is the pointer to some memory that the compiler doesn't know the type of"

- Any pointer to object can be assigned to a **void***

  - **int* pi = new int;**
  - **double* pd = new double[10];**
  - **void* pv1 = pi;**
  - **void* pv2 = pd;**

void* is useful for copying -

- To use a **void*** we must tell the compiler what it points to

```
void f(void* pv)
{
    void* pv2 = pv;    // copying is ok (copying is here)
    double* pd = pv;  // error: can't implicitly convert void* to double*
    *pv = 7;          // error: you can't dereference a void*
                      // good! (The int 7 is not represented  like the double 7.0)
    pv[2] = 9;            // error: you can't subscript a void*
    pv++;             // error: you can't increment a void*
    int* pi = static_cast<int*>(pv);      // ok: explicit conversion
    // ...
}
```

- A **static_cast** can be used to explicitly convert to a pointer to object type – not recommended

Pointers and references

- A reference is an automatically dereferenced pointer
  - or can be thought of as “an alternative name for an object”
  - A reference must be initialized
  - **The value of a reference cannot be changed after initialization**

```
int x = 8;
int y = 9;
int* p = &x;    *p = 10;
p = &y;      // ok
int& r = x;x = 11;
r = &y;// error
```

VECTORS, TEMPLATES AND EXECEPTIONS

PART 6

Overview

- Vector definitions - How are they implemented?
- Pointers and memory - the free store
- Initialization and Destructors
- Copy and move

- Changing size
  - resize() and push_back()
- Templates
- Range checking and exceptions

Changing vector size – in three ways

- Given

  vector v(n); *// v.size()==n*

The three ways are:

- Resize it directly:
  - **v.resize(20);** *// v now has **20** elements*
- Add an element
  - **v.push_back(8);** *// add an element with value 8 to the end of **v***

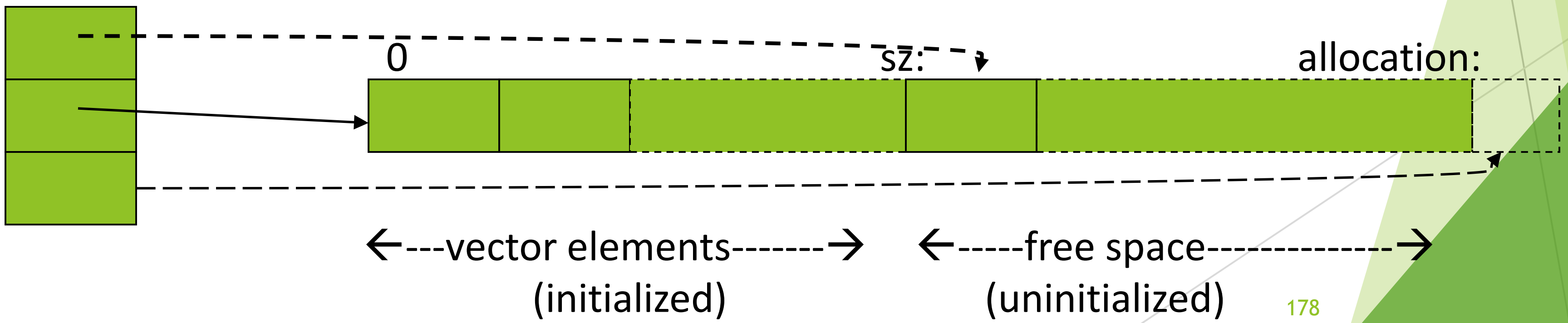
    *// **v.size()** increases by **1***
- Assign to it
  - **v = v2;** *// v is now a copy of **v2***

    *// **v.size()** now equals **v2.size()***

Representing vector - resize() or push_back() operations

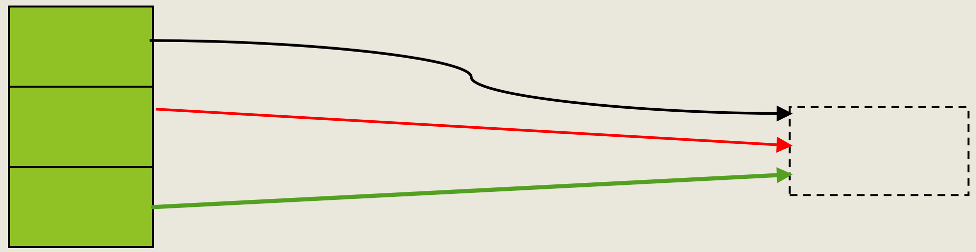
- Preparing the free space for future expansion

```
class vector {
      int sz;
      double* elemt;
      int space;    // total number of elements and the “free space”
                // the number of  spaces free  for the new elements
public:
      // …
};
```

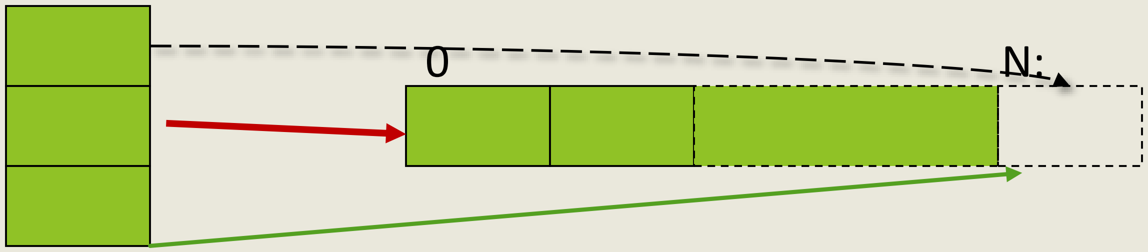


Vectors

- An empty vector with free store use:



- A vector(n) - with no free space:



vector::reserve()
used for new space allocation

- First deal with space (allocation); given space all else is easy
  - Note: **reserve()** doesn't assess the size or element values

```
void vector::reserve(int newspace)
    // make the vector have space for newspace number of elements
{
    if (newspace<=space) return;            // never decrease allocation

    double* p = new double[newspace];   // allocate new space
    for (int i=0; i<sz; ++i) p[i]=elemt[i];  // copy old elements
    delete[ ] elemt;                     // deallocate old space
    elemt = p;
    space = newspace;
}
```

vector::resize()

- Given **reserve()**, **resize()** is easy
  - **reserve()** does with space/allocation
  - **resize()** handles the element values

```
void vector::resize(int newsize)
    // make the vector have newsize elements
    // initialize each new element and each has a default value 0.0
{
    reserve(newsize);            // make sure we have sufficient space
    for(int i = sz; i<newsize; ++i) elemt[i] = 0;     // initialize new elements
    sz = newsize;
}
```

vector::push_back()

- Given **reserve()** deals with space/allocation
  - **push_back()** just adds a value

```
void vector::push_back(double x)
    // increase vector size by one
    // and then initializes the new element with a
{
    if (sz==0)              // no space   - make some
            reserve(10);
    else if (sz==space)     // no more free space: get more space
     reserve(2*space);
    elemt[sz] = x;          // add a at end
    ++sz;             // and increase the size (sz is the number of elements)
}
```

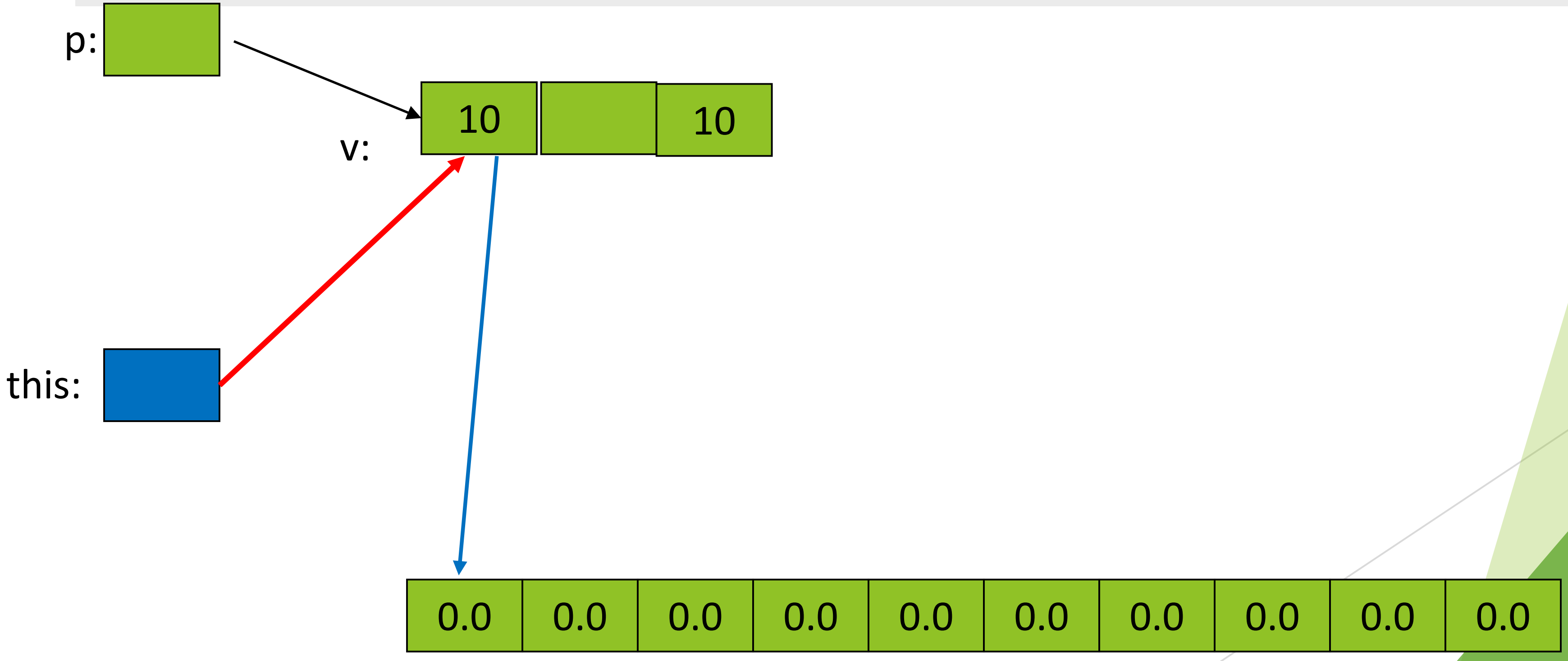
resize() and push_back()

```
class vector {      // a vector of doubles
    int sz;             // the size
    double* elemt;              // a pointer to the elements
    int space;              // size+free_space
public:
    // ... constructors and destructors ...

    double& operator[ ](int n) { return elemt[n]; }  // access: return reference
    int size() const { return sz; }                  // current size

    void resize(int newsize);                        // make bigger
    void push_back(double x);                 // add  in  element

    void reserve(int newspace);                      // get more space
    int capacity() const { return space; }           // current available space
};
```

The this pointer

- A vector is an object – for example **vector v(10);**
  - **vector* p = &v;**// *we can point to a* ***vector*** *object*

- When a, **vector**’s member functions need to refer to that object
  - “pointer to self” in a member function is **this**



The this pointer

```
vector& vector::operator=(const vector& a)
    // like copy constructor that first deals with old elements
{
    // …
    return *this;// by convention,
           // assignment returns a reference to its object: *this
}

void f(vector v1, vector v2, vector v3)
{
    // …
    v1 = v2 = v3;      // rare use made possible by operator=() returning *this
    // …
}
```

- The **this** pointer has more uses … see next slides

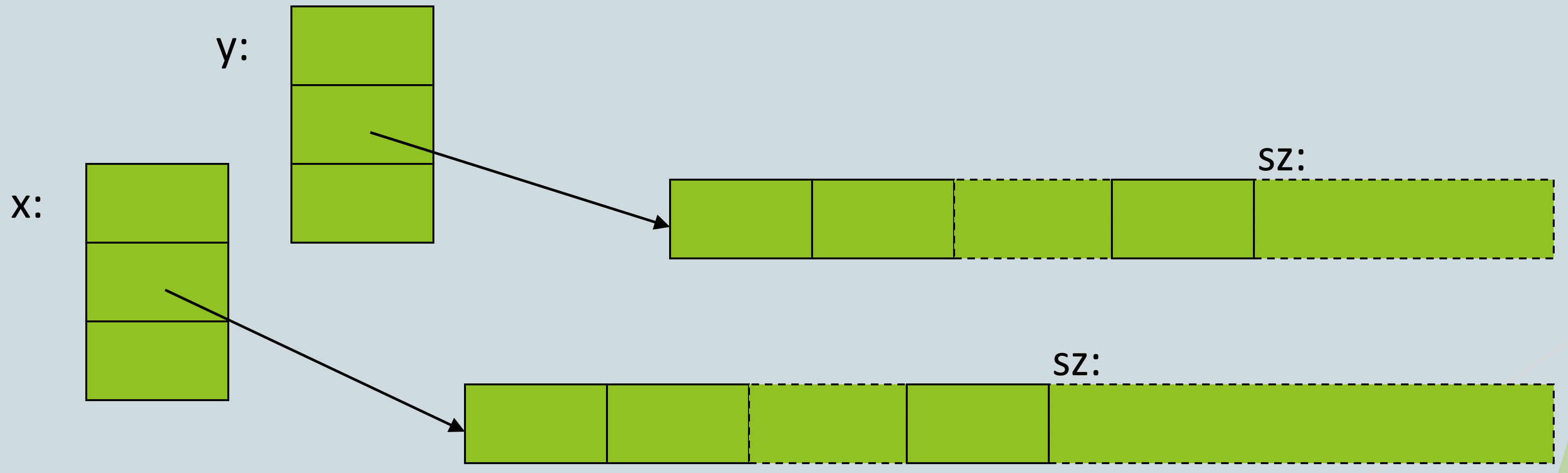
Copy and swap with vectors

- Copy and swap is important idea:

```
vector& vector::operator=(const vector& a)
    // like copy constructor, ---  but first must deal with old elements
    // make a copy of a then replace the current sz and elemt with a's
{
    double* p = new double[a.sz];                // allocate new space
    for (int i = 0; i<a.sz; ++i) p[i] = a.elemt[i]; // copy elements
    delete[ ] elemt;                     // deallocate old space
    sz = a.sz;                           // set new size
    elemt = p;                           // set new elements
    return *this;            //  return a  this pointer
}
```

Sufficient space in the target vector

- “Copy and swap” is but not always the most efficient
  - What if there already is sufficient space in the target vector?
    - Then just copy
    - For example: **x = y;**



Optimized copy and swap

```
vector& vector::operator=(const vector& a)
{
    if (this==&a) return *this;     // self-assignment, no more to be done
ore
    if (a.sz<=space) {          // enough space, no need for new allocation
      for (int i = 0; i<a.sz; ++i) elemt[i] = a.elemt[i];     // copy elements
      space += sz-a.sz;                          // increase free space
      sz = a.sz;
      return *this;
    }

    double* p = new double[a.sz];                   // copy and swap
    for (int i = 0; i<a.sz; ++i) p[i] = a.elemt[i];
    delete[ ] elemt;
    sz = a.sz;
    space = a.sz;
    elemt = p;
    return *this;
}
```

Templates

- We will explain templates with the example of vectors
- with element types we specify
  - **vector<double>**
  - **vector<int>**
  - **vector<Month>**
  - **vector<Record*>** *// vector of pointers*
  - **vector<vector<Record>>** *// vector of vectors*
  - **vector<char>**

- We make the element type a <u>parameter</u> to **vector**

- The Templates enable the creation of a class can have the type of things it works on to be changed

Templates

- Provides for ‘generic programming in C++’
    - Parameterization of types (and functions) by types (and integers)
  - Very good for flexibility and performance
    - Used where performance is essential (*e.g.*, real time and numerical calculations) and where flexibility is essential (*e.g.*, the C++ standard library)
- Template definitions

  template<class T, int N> class Buffer { /* … */ };
  template<class T, int N> void fill(Buffer<T,N>& b) { /* … */ }

- Template specializations (instantiations)

  // for a class template, one must specify the template arguments:
  Buffer<char,1024> buf; *// for **buf**, **T** is **char** and **N** is **1024***

  // for a function template, simply provide the template arguments:
  fill(buf); *// for **fill()**, **T** is **char** and **N** is **1024**; that's what **buf** has*

Templates – a class is a type so declare variables of that type

- Class PETROL_STATION has three instantiated variables – MAXOL, TEXACO, MOBILE

```
class PETROL_STATION
   {   public
          int first;
          int second;
          int third;
          int sum()
            {    return first + second + third;
             }
   };
```

- At run-time the instances of class PETROL_STATION are created, then call sum()

Parameterize with element type

```
// an almost real vector of Ts:
template<class T> class vector {
    // …
};
vector<double> vd;          // T is double
vector<int> vi;             // T is int
vector<vector<int>> vvi;    // T is vector<int>
                            //    in which T is int
vector<char> vc;            // T is char
vector<double*> vpd;        // T is double*
vector<vector<double>*> vvpd;   // T is vector<double>*
                                // in which T is double
```

Basically, **vector<double>** is *a* ***vector of doubles:***

```
class vector {
    int sz;             // the size
    double* elemt;      //  pointer for the elements
    int space;          // size+free_space
public:
    vector() : sz(0), elemt(0), space(0) { }            // default constructor
    explicit vector(int s) :sz(s), elemt(new double[s]), space(s) { } // constructor
    vector(const vector&);                 // copy constructor
    vector& operator=(const vector&);          // copy assignment
    ~vector() { delete[ ] elemt; }              // destructor

    double& operator[ ] (int n) { return elemt[n]; }      // access: return reference
    int size() const { return sz; }                // the current size

    // …
};
```

Vector for <char>

```
// a vector of chars:
class vector {
    int sz;            // the size
    char* elemt;       //  pointer for the elements
    int space;         // size+free_space
public:
    vector() : sz{0}, elemt{0}, space{0} { }          // default constructor
    explicit vector(int s) :sz{s}, elemt{new char[s]}, space{s} { } // constructor
    vector(const vector&);                    // copy constructor
    vector& operator=(const vector&);          // copy assignment
    ~vector() { delete[ ] elemt; }              // destructor

    char& operator[ ] (int n) { return elemt[n]; } // access: return reference
    int size() const { return sz; }             // the size

    // …
};
```

The vector<T> is

```
// a vector of Ts:
template<class T> class vector {      //designed for "for all types T
     int sz;              // the size
     T* elemt;            // the pointer to the elements
     int space;           // size+free_space
public:
     vector() : sz{0}, elemt{0}, space{0};        // default constructor
     explicit vector(int s) :sz{s}, elemt{new T[s]}, space{s} { }    // constructor
     vector(const vector&);                  // copy constructor
     vector& operator=(const vector&);            // copy assignment
     vector(const vector&&);                  // move constructor
     vector& operator=(vector&&);             // move assignment

     ~vector() { delete[ ] elemt; }                  // destructor


     // ...
};
```

The vector<T> is

```
// the vector of Ts:
template<class T> class vector {      // designed  "for all types of T"
    int sz;             // the size
    T* elemt;           // for the pointer to the elements
    int space;          // size+free_space
public:
    // … constructors and destructors …

    T& operator[ ] (int n) { return elemt[n]; }       // access: return reference
    int size() const { return sz; }                    // the current size

    void resize(int newsize);                  // grow
    void push_back(double d);                  // add element

    void reserve(int newspace);                // get more space
    int capacity() const { return space; }       // current available space

    // …
};
```

Error Handling - Exceptions

- Use exceptions to report errors
- We must ensure that use of exceptions
  - Doesn't become a new sources of bugs
  - Doesn't complicate the code
  - Doesn't lead to memory leaks

Resource management

- A resource is something that has to be acquired and must be released properly
- Examples of resources
  - Memory
  - Locks
  - File handles
  - Thread handles, sockets

```
void resource_mgmt._issue (int s, int x)
{
    int* p = new int[s];    // adding memory
    // . . .
    delete[] p;             // releasing memory
}
```

Resource management

- Why difficult?
  - It is easy to make mistakes with pointers and delete

```
void resource_mgmt_issue(int s, int x)
{
        int* p = new int[s];        // acquire memory
        // . . .
        if (x) p = q;               // then  p is made point to another object
        // . . .
       delete[] p;            // release memory --but the wrong memory
}
```

Resource management

- Why Difficult- issue may be that we did not get to the end of the function :
  - It's easy to make this mistake…

```
void resource_mgmt._issue(int s, int x)
{
        int* p = new int[s];      // acquiring the memory
        // . . .
      if (x) return;          // So if we don't get to the end: leak
       // . . .
       delete[] p;             // release memory
}
```

Resource management

- Why Difficult?
  - Again…if not to get to the end of the function

```
void resource_mgmt._issue (int s, int x)
{
        int* p = new int[s];        // acquiring the memory
        // . . .
      if (x) p[x] = v[x];           // v[x] may throw an exception: leak
       // . . .
       delete[] p;       // releasing the memory
}
```

Resource management

- Rudimentary fix code:

```
void resource_mgmt._issue(int s, int x)  // difficult code
{
        int* p = new int[s];        // acquiring the memory
        vector<int> v;
        // . . .
        try {
                if (x) p[x] = v[x]; // may throw the exception
                // . . .
        } catch (. . .) {           // catching every exception
                delete[] p;         // releasing the memory
                throw;              // re-throwing an exception
        }
        // . . .
        delete[] p;                 // release memory
}
```

Resource management

- Simple Approach
  - RAII: “Resource Acquisition is initialization”
  - Vector’s destructor releases memory upon scope exit

```
void f(vector<int>& v, int s)
{
        vector<int> p(s);
        vector<int> q(s);
        // . . .
}  // destructor releases memory upon scope exit
```

Resource management

- But what about functions creating objects?
  - The error-prone solution: return a pointer

```
vector<int>* make_vec()          // make a filled vector
{
        vector<int>* p = new vector<int>;   // allocate on free store
     // ...assign vector with data;…. Notice if this will throw an exception . .
        return p;
}
```

*// now users have to remember to **delete***

// this will lead to a memory leak!

Resource management

- But what about functions creating objects?

  - Improved solution: use **std::unique_ptr**

```
unique_ptr<vector<int>> make_vec()        // make a filled vector
{
        unique_ptr<vector<int>> p {new vector<int>};   // allocate on free store
        // … fill the vector with data; this may throw an exception …
        return p;
}
```

*// users don't have to **delete**; no **delete** in user code*

*// a **unique_ptr** owns its object and deletes it automatically*

Resource management - std::make_unique C++14 used

- But what about functions creating objects?
  - Even better solution: use **std::make_unique**
  - C++14 only (unless you have an implementation of **make_unique**)

```
unique_ptr<vector<int>> make_vec()        // make a filled vector
{
        auto p = make_unique{vector<int>};  // allocate on free store
        // ... fill the vector with data; this may throw an exception ...
        return p;
}

// no new in user code
```

Resource management - objects

- But what about functions creating objects?
  - Best solution – don't use pointers at all
  - Instead return the object itself

```
vector<int> make_vec()        // make a filled vector
{
        vector<int> res;
        // . . . fill the vector with data; this may throw an exception . . .
        return res;     // vector's move constructor efficiently transfers ownership
}
```

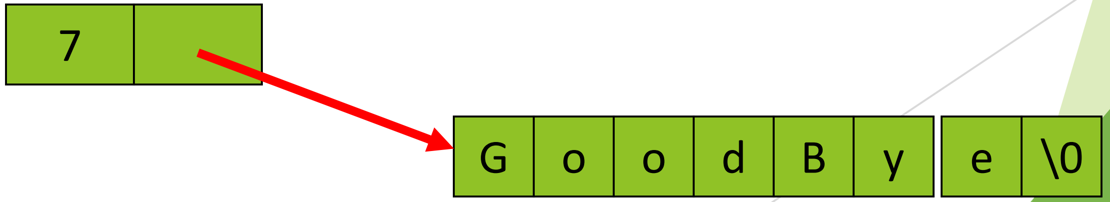
Resource Acquisition is Initialization (RAII)

- Also called "**Scoped Resource Management**"

- Vectors
  - acquires memory in its constructor
  - Gives back (releases) the memory in the destructor
  - Is simpler and works well with error handling with exceptions
  - Examples - Memory, file handles, sockets, I/O connections - iostreams handle those using Resource Allocation RAII), locks, widgets, threads.

String

- A **string** is very similar to a vector<char>
  - E.g. has the use of **size(), [ ], push_back()**
  - Built with the same language features and techniques
- BUT a **string** is optimized for character string manipulation
  - That is Concatenation (**+**)
  - Similar to the C-style string (**c_str()**)
  - >> input terminated by whitespace
  - Small strings don't use free store (instead are stored in the handle)



STANDARD TEMPLATE LIBRARY – LINKED LISTS

PART 8

Motivation

- A “List” is a useful structure to hold a collection of data.
  - It is heavily used in the ‘real world’
- Examples:
- A list of images to be burned to a CD in a medical imaging application
- A list of users of a website that need to be emailed some notification
- A list of objects in a 3D game that need to be rendered to the screen

Using arrays - we need to oversize in advance

- list using static array -

```
int my_Array[100];
int n;
```

Using dynamic array - We allocate an array (list) of any specified size while the program is running

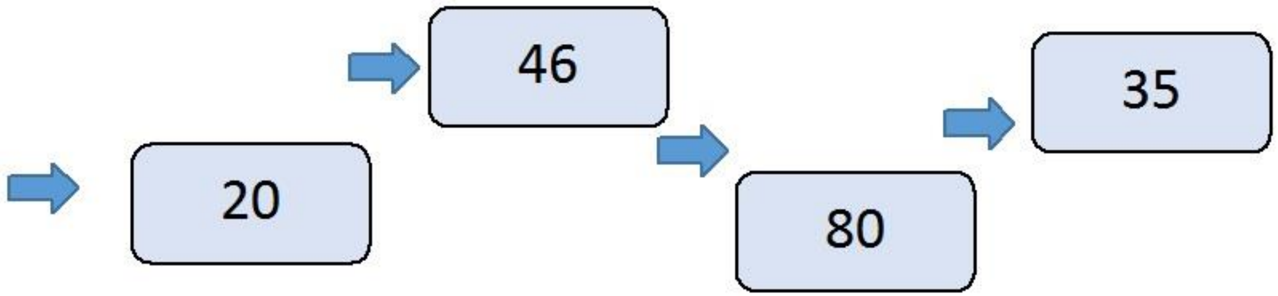
```
int* my_Array;
int n;
cin >> n;
my_Array = new int[n];
```

BUT linked-list (dynamic size)

size = ??

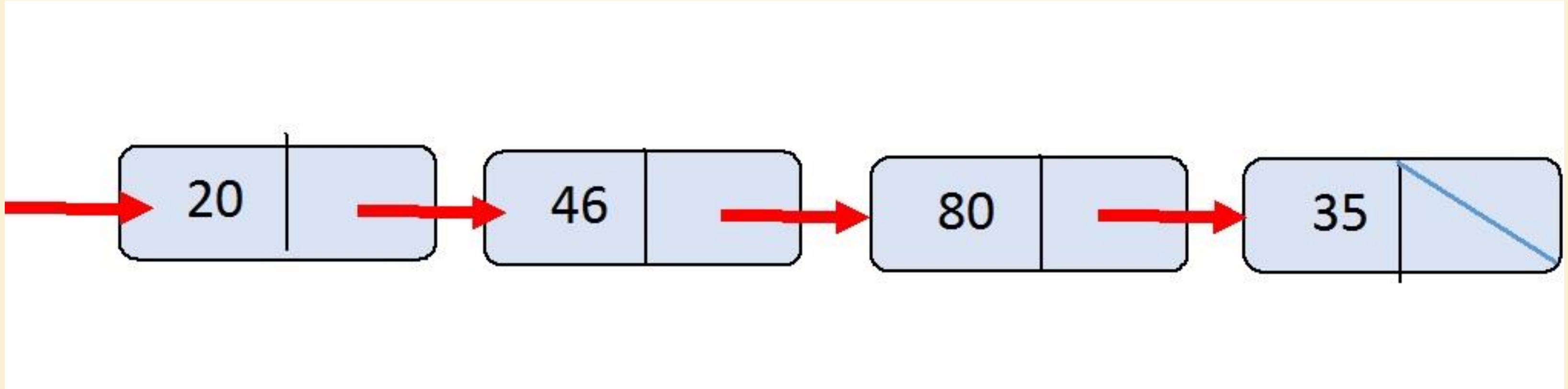
The list is dynamic, so it can grow and shrink to any size.

Array naturally represents an ordered list while the link list is implicit, consecutive and contiguous



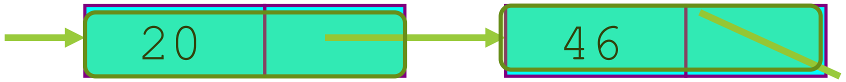
Linked Lists: Basic Idea – data with links

- A linked list is an *ordered* collection of data
- Each element of the linked list has
  - data
  - A link to the next element
- The link is used to chain the data eg. e linked list of integers:

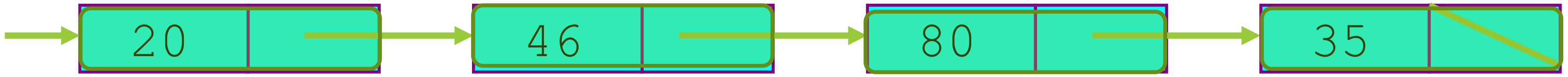


Basic Ideas

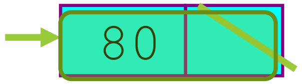
- The list can grow and shrink



addEnd(80), addEnd(35)

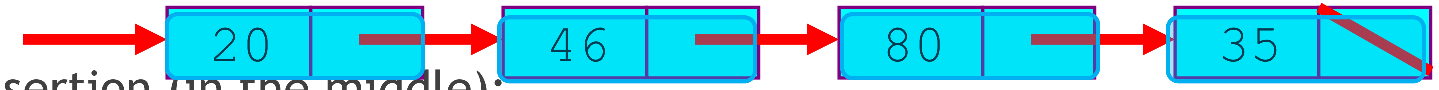


deleteEnd(35), delete_Head(20), delete_Head(46)

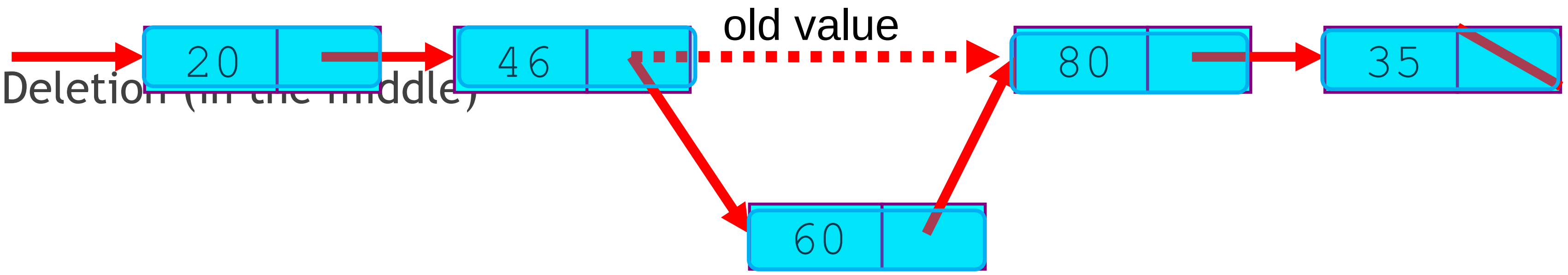


Linked Lists: Operations

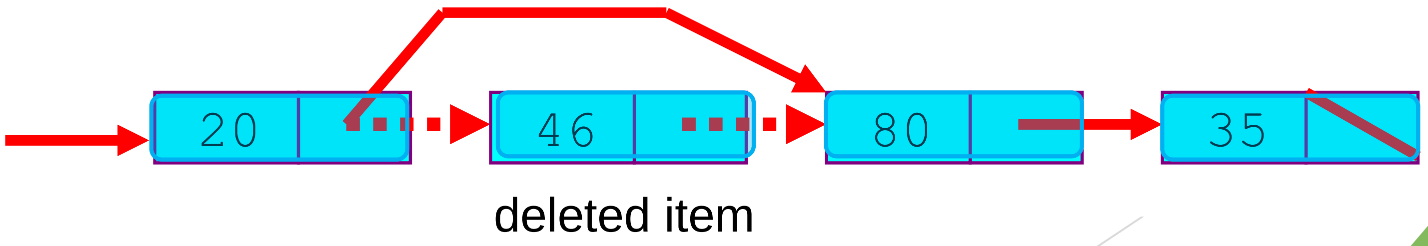
- Original linked list of integers:



- Insertion (in the middle):



- Deletion (in the middle)



Link list type definition

```
struct Node{
  int data;
  Node* next;
};

typedef Node* Node_Ptr;
```

typedef for node definition

```
typedef int WARH;
WARH k;    // same as:  int k;

typedef int* WAH_PTR;
WAH_PTR p; // same as: int* p;

typedef Node* Node_Ptr;
Node_Ptr Head;  // same as: Node* Head;
```

Linked List Structure

- Definition

```
struct Node {
    int data;
    Node* next;
};
```

- Create a Node

```
Node* p;
p = new Node;
```

- Delete a Node

```
delete p;
```

- Definition

```
struct Node {
    int data;
    Node* next;
};

typedef  Node*  Node_Ptr;
```

- Create a Node

```
Node_Ptr p;
p = new Node;
```

- Delete a Node

```
delete p;
```

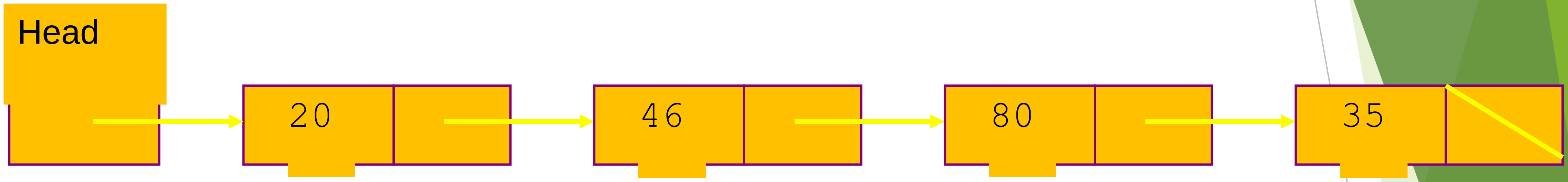
To access fields in a node

```
(*p).data;  //access the data field
(*p).next;  //access the pointer field
```

Alternatively - it can be accessed this way:

```
p->data        //access the data field
p->next        //access the pointer field
```

Representing and accessing linked lists



We define a pointer to the start:

```
Node_Ptr head;    //(also: Node* head)
```

that points to the first node of the linked list, when the linked list is empty then head is NULL.

Linked Lists - passing to a Function - very similar to an array

- When passing a linked list to a function - all that is needed is to pass the value of `head`.
- Then we can use the pointer to or the value of `head` the function can access the entire list.

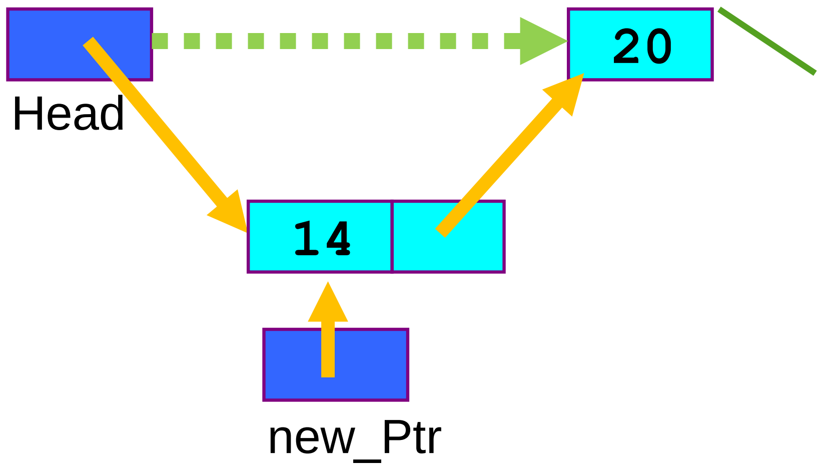
- Problem: What if the function changes the beginning of a list by inserting or deleting a node, then `head` will no longer point to the beginning of the list?
- Solution: Pass it by reference - when passing `head` always

  to the function to return a new pointer value

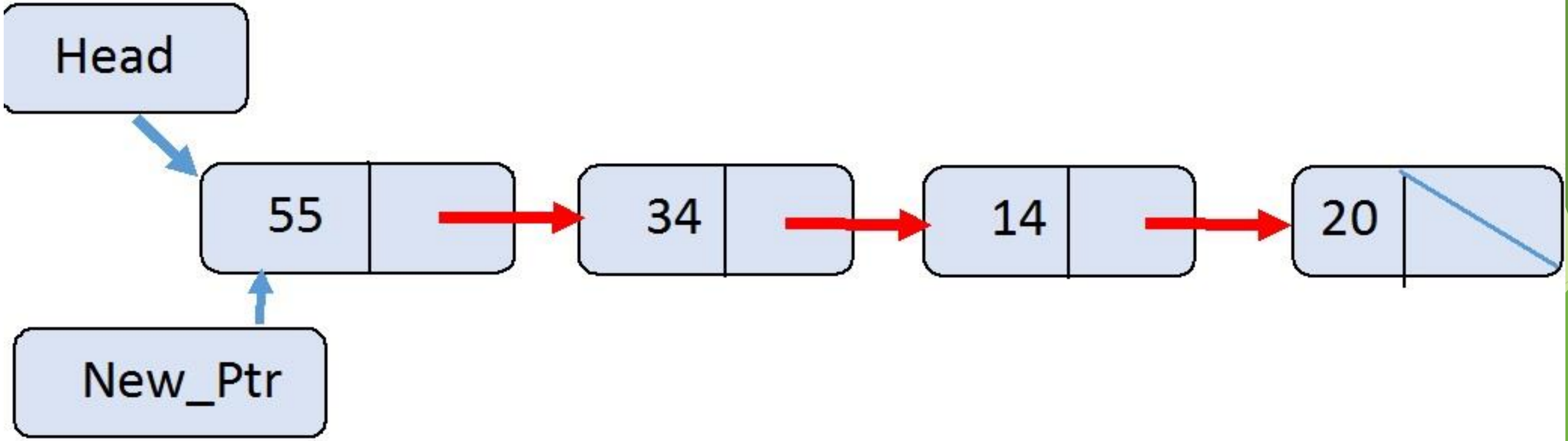
Implementation of a Linked List - Unsorted

Inserting a Node at the Beginning

```
new_Ptr = new Node;
new_Ptr->data = 14;
new_Ptr->next = Head;
head = new_Ptr;
```



Insert a few more entries

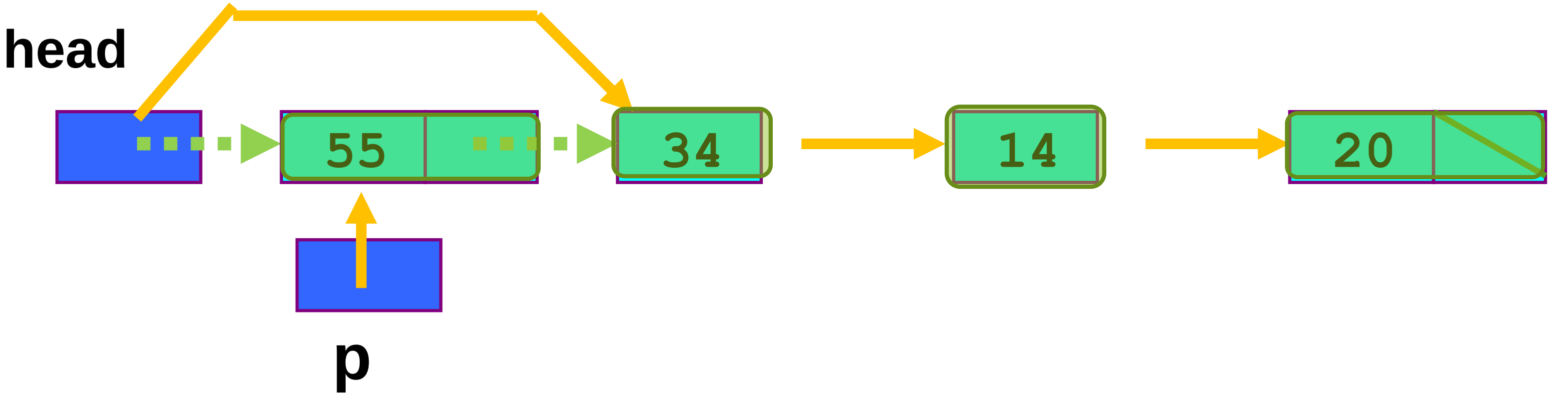


Add an element to the head:

```
Node_Ptr add_to_Head(Node_Ptr head, int
  new_data){

  Node_Ptr new_Ptr = new Node;

  new_Ptr->data = new_data;
  new_Ptr->next = Head;

  return new_Ptr;
}
```

Deleting the Head Node

```
Node_Ptr p;

p = head;
head = head->next;
delete p;
```



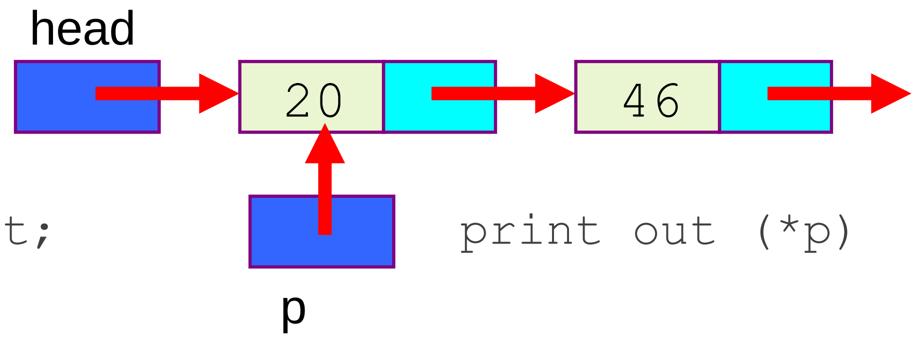
```
void delete_Head(Node_Ptr& head){

    if(head != NULL){
        Node_Ptr p = head;
            head = head->next;
            delete p;
    }
}
```

Or as the function returning the head pointer

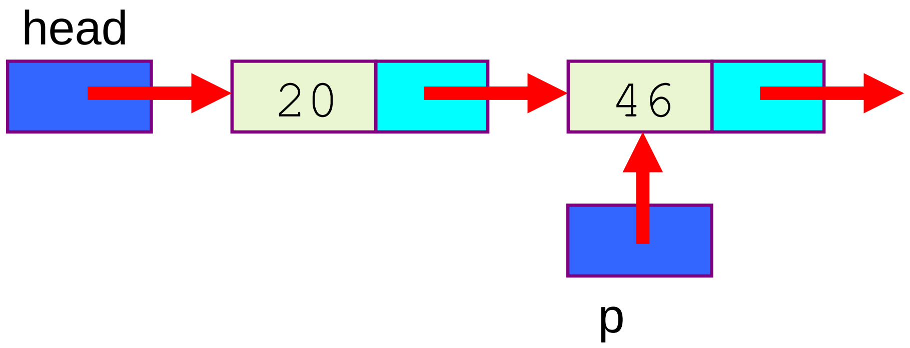
```
Node_Ptr delete_Head(Node_Ptr head){

    if(head != NULL){
        Node_Ptr p = head;
            head = head->next;
            delete p;
    }

    return head;
}
```

Displaying a Linked List - print out (*p)

p = head;   print out (*p)



p = p->next;   print out (*p)



A linked list is displayed by walking through its nodes one by one, and displaying their data fields - similar to an array

```
void display_List(Node_Ptr
 head){
    Node_Ptr p;
    p = head;
    while(p != NULL){
      cout << p->data << endl;
      p = p->next;
    }
  }
```

Searching for a value in a linked list - can look at array searching first!

Similar to an Array:

```
void display_Array(int
 data[], int size)  {
    int n=0;
    while ( n<size ) {
    cout << data[i] << endl;
    n++;
   }

}
```

```
void display_Array(int
 data[], int size)  {
    int* p;
    p=data;
    while ((p->data)<size ) {
    cout << *p << endl;
    p++;
   }

}
```

Remember searching algorithm for the place in an array is very similar

```
void main() {
    const int size=8;
    int data[size] = { 11, 17, 92, 6, 27, 32, 55, 9 };
     int value;
    cout << "Enter a search item: ";
     cin >> value;
    int n=0;
    int position=-1;
    bool found=false;
    while ( (n<size) && (!found) ) {
    if(data[n] == value) {
        found=true;
        position=n;}
        n++;
  }
  if(position==-1) cout << “We have not found it!!\n";
  else cout << “We found it at: " << position << endl;
}
```

Searching for a value in the list with arrays

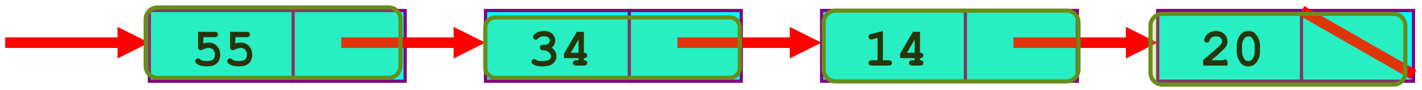
```
Node_Ptr search_Node(Node_Ptr
 head, int item){
   Node_Ptr p = head;
     Node_Ptr result = NULL;
   bool found=false;
   while((p != NULL) &&
   (!found)){
    if(p->data == item) {
        found = true;
        result = p;}
    p = p->next;
   }
   return result;
}
```

If we use a pointer to an array, it is very similar

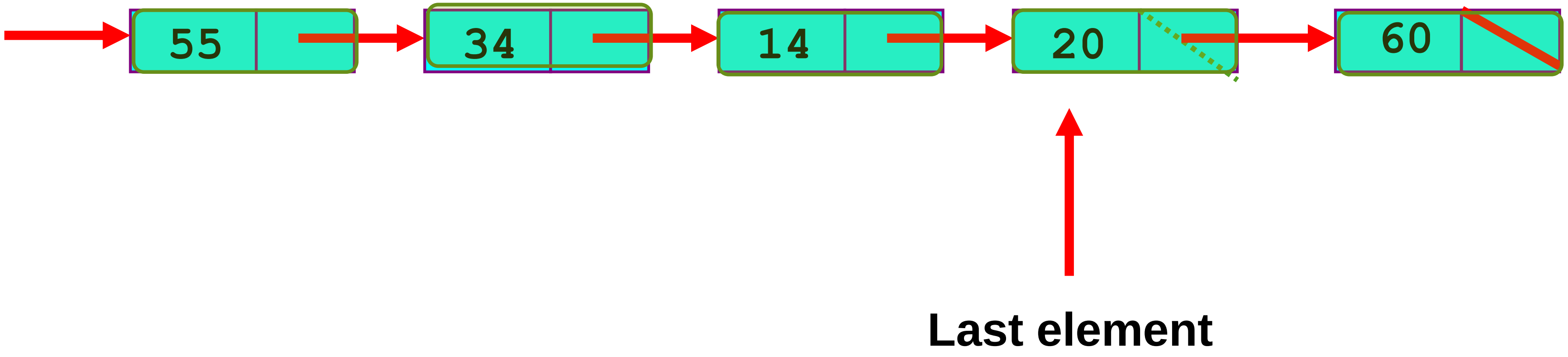
```
int search_Array(int data[], int size,
   int value){
    int n=0;
    int position=-1;
    bool found=false;
    while ( (n<size) && (!found) ) {
    if(data[n] == value) {
        found=true;
        position=n;}
            n++;
   }
   return position;
}
```

More operations - adding to the end

- The original linked list of integers was:



- So - to add to the end … insert:



The key is how to locate the last node of the list

Adding to the end:

```
void addEnd(Node_Ptr& head, int new_data){
   Node_Ptr new_Ptr = new Node;
   new_Ptr->data = new_data;
   new_Ptr->next = NULL;

   Node_Ptr last = head;
   if(last != NULL){  // general non-empty list case
    while(last->next != NULL)     //
        last=last->next;
    last->next = new_Ptr;        //
   }
   else   // deal with the case of empty list
    head = new_Ptr;
}
```

Link new object to last->next

Link a new object to empty list

Function for ‘Add to the end’

```
Node_Ptr addEnd(Node_Ptr head, int new_data){
   Node_Ptr new_Ptr = new Node;
   new_Ptr->data = new_data;
   new_Ptr->next = NULL;

   Node_Ptr last = head;
   if(last != NULL){  // the non-empty list case
    while(last->next != NULL)
        last=last->next;
    last->next = new_Ptr;
   }
   else// the case of empty list
    head = new_Ptr;

   return head;
}
```

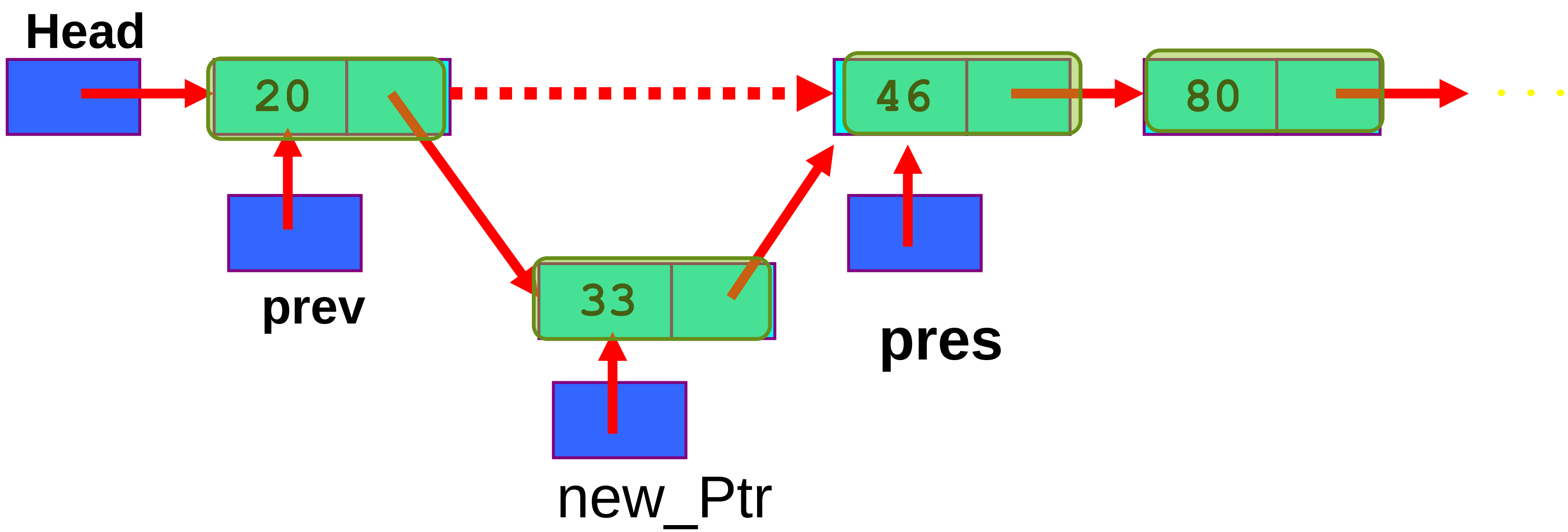
Implementation of a Linked List - Sorted

Inserting a value in a sorted list

How to do it in a sorted array?

1. Find the position
2. Free up the place by moving the other nodes
3. Insert the new value in the array or list

Inserting new node



Finding `prev` and `pres` nodes

Suppose that we want to insert or delete a node with data value `new_Value`.

Then the following code successfully finds `prev` and `pres` such that:

```
prev->data < new_Value <= pres->data
```

Search for value

```
prev = NULL;
pres = head;
found=false;

while( (pres!=NULL) && (!found) ) {
   if (new_Value > pres->data) {
      prev=pres;
      pres=pres->next;
   }
   else found = true;
}
```

A Boolean search clause - Logical AND (&&)

i.e. if the first part is false, the second part will not be executed.

```
prev = NULL;
pres = head;

while( (pres!=NULL) && (new_Value>pres->data) ) {
      prev=pres;
      pres=pres->next;
}
```

```
//insert item into linked list according to ascending order

void insertNode(Node_Ptr& head, int item){
   Node_Ptr newp, pres, pre;
   newp = new Node;
   newp->data = item;

   pre = NULL;
   pres = head;
   while( (pres != NULL) && (item>pres->data)){
    pre = pres;
    pres = pres->next;
   }

   if(pre == NULL){   //insert to head of linked list
    newp->next = head;
    head = newp;
   } else {
    pre->next = newp;
    new->next = pres;
   }
}
```

If the position happens to be at the head

The general case

Delete an element in a sorted list

```
void delete_Node(Node_Ptr& head, int item){
   Node_Ptr prev=NULL, pres = head;
   while( (pres!=NULL) && (item > pres->data)){
    prev = pres;
    pres = pres->next;
   }

if ( pres!==NULL && pres->data==item)   {
    if(pres==Head)
        Head = Head->next;
    else
        prev->next = pres->next;
    delete pres;
   }
}
```

Get the location of the element

If not found continue

Check I the element s present before delete

If (pres==NULL || pres->data!=item) Item is not in the list!

When the search is at the head

General delete case

Other variants of linked lists

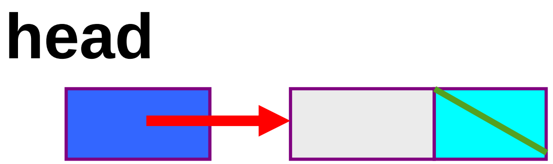
The Dummy Head node

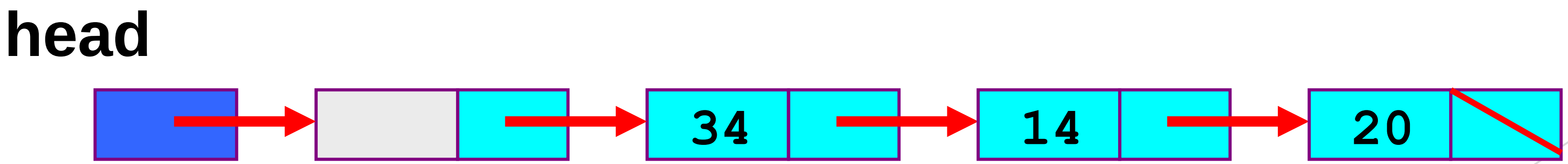
A well-known implementation ‘trick’ or ‘method’: add one more node at the beginning, which does not store any data, to ease operations.

- always present, even when the linked list is empty
- Insertion and deletion algorithms initialize `prev` to reference the dummy head node, rather than *NULL*

The ‘Dummy Head node’

- For empty lists:



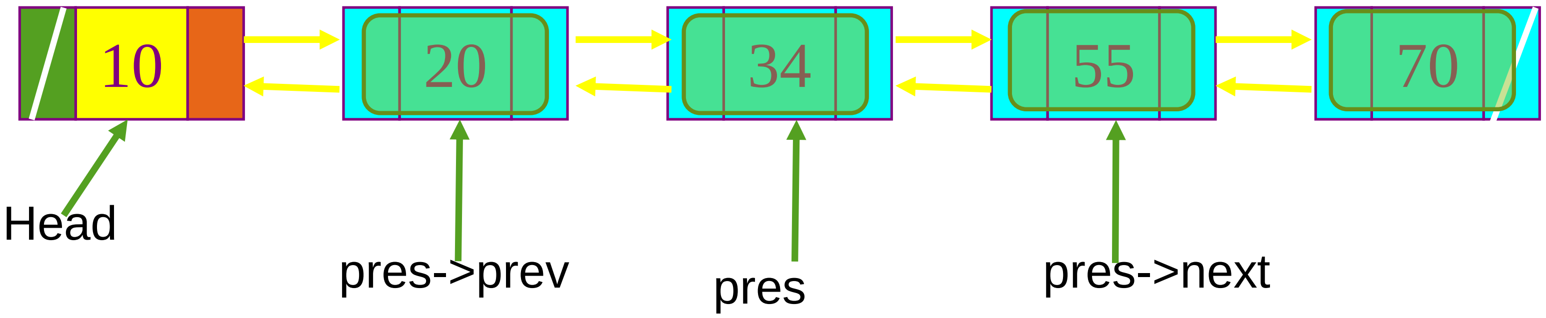


Dummy head node

Doubly Linked Lists

Doubly Linked-List means that each node has pointers pointing to both its predecessor and successor:

- ->prev points to the predecessor node
- ->next points to the successor node



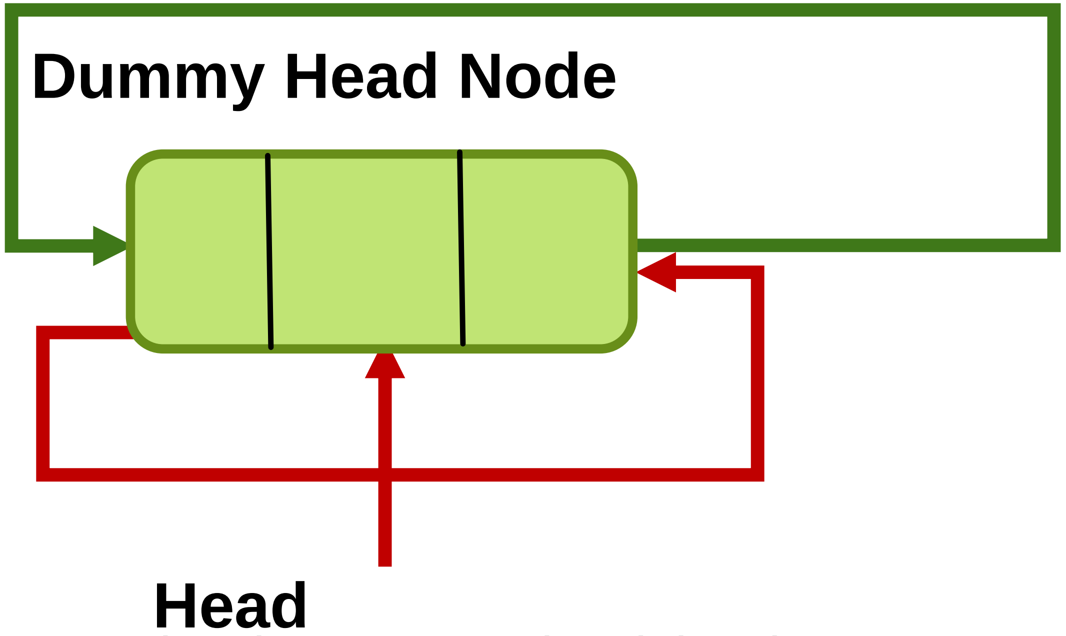
Doubly Linked List Definition

```
struct Node{
        int data;
        Node* next;
        Node* prev;
};
typedef Node* Node_Ptr;
```

Doubly Linked Lists
with Dummy Head Node - simplify insertion and deletion

- a dummy head node is added at the head of the list - to simplify insertion and deletion - by avoiding special cases of deletion and insertion at front and end
- The last node also points to the dummy head node as its next node

Empty List



Head->next = head;   compared with head=NULL;

Binary tree

A binary tree with two pointers …

```
Struct BinaryNode {
  double element; // the data
  BinaryNode* left; // left child
  BinaryNode* right; // right child
}
```

Associative Containers and Sequential Containers

- SET stores unique values sorted on insertion
- Map stores key-value pairs sorted by unique keys

- Sequential Containers we have already seen:
- - vector – operates like a dynamic array and grows at the end
- - deque –similar to vector except allows for elements to be inserted and removed from the beginning as well as the end
- - list can remove links from any position easily

MAPs and SETs

Containers

- STL has two kinds of containers
- Associative Containers
- Sequential Containers

- MAP is a associative container type

MAPs

- MAPs provide the CONTAINER type class from the standard template library
- Used for QUICK SEARCHS
- Each member of the MAP has a KEY and a VALUE, with a PAIR relationship
- Map <key Type, Value Type> mapObject
- This is a ‘PAIR’ template

Using the iterator to keep track of each word and how often we see it

```
int main()
{
 map<string, int>  COUNTERS;
// read the input and keep track of each word and how often we see it
 while(cin>>s)  ++ COUNTERS[s];

// write out the words and the associative counts
 for map<string, int> :: const_iterator it = COUNTERS.begin();
  {
  (it!=  COUNTERS.end(); ++ it)
   cout << it->first <<  \t  <<it->second << endl;
  }
return 0;
}
```

Code example

- #include <iostream>
- #include <map>
- #include <vector>
- Using namespace std;

- int main()
- map<string,int> NumberWords;
- NumberWords[“ten”] = 10;
- NumberWords[“twenty”] = 20;
- NumberWords[“thirty”] = 30;
- map<str,ing,int>::iterator loopy = NumberWords.begin();

Map has an iterator called ‘loopy’

```
map<string,int>::iterator loopy = NumberWords.begin();
while (loopy!NumberWords.end()
  {
     cout << loopy ->  first  <<  “ “;
      cout << loopy-> second <<  endl;
     loopy++
    }
```

Associative Data Structure

- **Lets us insert and find elements quickly according to their key**
- **Map is the same as vector but does not need to be an int**
- **And has *an iterator built in***

Instantiating MAP and int Key to a String Value

- #include <map>
- #include<string>
- Template <typename KeyType>
- struct ReverseSort
- { bool operator() (const KeyType& key1, const KeyType& key2)

- int main()
- { using namespace std;
- //map key of type int to value of type string
- map<int, string> mapIntToString1;
- //map constructed as copy of another
- map <int, string> mapIntToString2, mapIntToString1);
- // map constructed given part of another map
- map <int,string> mapIntToString3(mapIntToString1.cbegin(); mapIntToString1.cend();
- //map with inverses a sort order
- map<int, string, ReverseSort <int>mapIntToString4
- (mapInttoString1.cbegin(), mapIntToString1.cend();
- Return 0;
- }

Associative Containers
SETS

Different Instantiation Techniques of Sets

- Template <typename T>
- struct SortDescending
- {
- bool operator() (const T& lhs, const T& rhs) const;
- {   return, (lhs>rhs);   }

//keep track of each word and how often we see it

- Int main()
- {   using namespacestd;
- set <int> setIntegers1;
- // set instantiated given a user-defined predicate
- set <int, SortDecending <int> msetInteger2
- // creating one set from another or part of one
- set <int> setIntegers3(setIntegers1);
- return 0;
- }

Instantiating a SET Object

- Set <int> setIntegers
- Set <TUNA> setIntegers;

- To declare an iterator to point to the element of the set

- Set <int>:: const iterator; iElementInSet
- multiset <int> :: const iterator iElementInSetMultiset;

Use as template parameter in set

- Template <typename T>
- Struct SortDescending
- {
- bool operator()
- (const T& lhs, const T& rhs) cont
- {
- return(lhs,rhs)
- }
- };

ADVANCED TOPICS OF C++ PROGRAMS

PART 10

Generic algorithms and Lambda functions in C++

Summary

- Some generic algorithms provided by the library
- lambda functions

Introduction

- The standard library provides more than 100 algorithms
- All of them can be very useful
- We will focus here in explaining some algorithms

The find() algorithm - a simple useful algorithm

- Starting with :

  find()

--- Suppose we have a vector of *ints* and we want to know if that vector holds a particular value?

```
int val = 56;                    // value we are searching for

//result will denote the element we will find if it’s in vec, or vec.cend() if it isnt

auto result=find(vec.cbegin(),vec.cend(),val);

cout << “The value “<< val  (result == vec.cend() ? “ is not found” : “ is found”) << endl;
```

- The first two elements are the iterators for a range of elements, and the third argument is the value
- It returns an iterator to the element or the second iterator if the element is not in the container

The find() algorithm

```
template<class In, class T>
In find(In first, In last, const T& val)
{
        while (first!=last
        && *first!=val)
        ++first;  return
        first;
}
```

The find_if() algorithm

```
template<class In, class Pred>
In find_if(In first_01, In last_01, Pred pred_01)
{
	while (first_01!=last && !pred_01(first_01))
	++first_01;
	return first_01;
}
```

Useful if the first element is to meet any criterion

we pass the function a *predicate* instead of a value

- Definition - A predicate is a function that returns true or false
- We have this because

find_if() requires a predicate that takes one argument

The find_if() algorithm

- We can use find_if() to find the first element which is odd in a sequence of ints

```
bool odd(int x) {return x%2:}
void f(vector<int> &v)
{
        vector<int>::iterator p = find_if(v.begin(), v.end(), odd);

        if (p!=v.end()) { // … there is an odd number, pointed by
        p
}
```

Or find the first element which is bigger than 56:

- Or find the first element which is bigger than 56?

```
bool larger_than_56(int x)
 {return x>56;}

void f(vector<int> &v)
  {
        vector<int>::iterator p = find_if(v.begin(), v.end(), odd);
         if (p!=v.end()) { // … we found a value bigger than 56?
```

- Why if we want to find any value? The first value biggest than 55?

The find_if() algorithm

- We can use find_if() to obtain the first element which is odd in a sequence as below:

```
bool odd(int x) {return x%2:}
void f(vector<int> &v)
{
        vector<int>::iterator p = find_if(v.begin(), v.end(), odd);
        if (p!=v.end()) { // … there is an odd number, pointed by p
}
```

The find_if() algorithm (cont)

- Or use the find for the first element which is bigger than 56

```
bool larger_than_56(int x) {return x>56;}  void
f(vector<int> &v)
{
        vector<int>::iterator p = find_if(v.begin(), v.end(), odd);
        if (p!=v.end()) { // … we found a value bigger than 56?
```

- Why if we want to find any value if the first value biggest than 55?

The find_if()   (cont)

- If we want to compare with v, we make v an implicit argument of the function will call -

```
double v_val;
bool larger_than_v(double x) { return x > v_val; }

void f(vector<int> &v)
{
        v_val = 55;
        vector<int>::iterator p = find_if(v.begin(), v.end(), larger_than_v);
        if (p!=v.end()) { // … there is an odd number, pointed by p
        }
```

- Maintaining this code is difficult, so there should be another way of doing so

Operator overloading

- In C++ we can define the meaning of an operator when applied to operands of a class type

  - For example, we can define the addition operator + for objects of a given type to allow for the addition of some its members

- Overloaded operators are functions with special names - keyword operator followed by the symbol of the operator being defined
  - Like any function, an overloaded operator has a return type, a parameter list, and a body
  - The number of arguments is the number of operands that the operator has

Function objects

- In our previous example what we want is a predicate to compare elements to a value that somehow we specify as some kind of argument
- Somehow we want something like this

```
void f(list<double>& v, int x) {
        list<double>::iterator p = find_if(v.begin(), v.end(), larger_than(31);
        }
```

- We can do this using a function object, i.e., an object that can behave like a function

We can do this using a function object, i.e., an object that can behave like a function

```
class larger_than {
        int v;
       public:
           larger_than(int vv) : v(vv) {};  //store the argument
                                            // return x>v;
           bool operator()(int x) const {return x>v;}
}
```

Note: **function objects are more efficient that passing a function some times, since the compiler is able to generate optimal code for that in many cases**

Introduction to lambda functions

- A lambda expression has the following form:
- **[*capture list*]** (*parameter list*) -> return type { *function body* }
- **capture list** is a list of local variables defined in the ‘square brackets’ function
- return type, parameter list and function body are the same as in any ordinary function

Introduction to Lambda functions

- we can omit the parameter list and return type,
- but we have to always include the capture list (which may be empty)
- andthe function body:

```
auto f = [] { return 56;};

cout << f() << endl; //prints 56
```

Passing arguments to a lambda

- As with usual function calls, the arguments that are passed to a lambda function are used to initialize the lambda parameters
- The arguments and the parameter types must match
- The lambda does not have default arguments
- The parameters are first initialized and then the body of the function is executed

Using the **capture** list

- A lambda function can use variables local to the function where it has been defined only if it mentions these variables in the **capture** list
- Looking back to the slides of the **find_if** function, to find the first element that is bigger than v, we can write it using a lambda function like:

```
v = 56;
auto p = find_if(vec.begin(), vec.end(), [v] (const int &a)
  {return a > v;});
```

Lambda captures and returns

- The local variables that are going to be used by our lambda

function using the capture list [&] or [=]

  - The first indicates that these variables are passed as references
  - The second indicates that these variables are passed by value

Lambda captures and returns

- By default, a lambda may not change the value of a variable, instead it copies by value
- Hence if we want to be able to change it, we need to use the keyword mutable after the
- parameter list

```
void function {
        int v1 = 56; // local variable
        auto f = [v1] () mutable
        {return ++v1;};
        v1 = 0;
        auto j = f(); // j has the value 43
}
```

Lambda captures and returns (cont).

- Variables captured by reference can be changed at any time (if not const)

```
void function {
        int v1 = 56; // local variable
        auto f = [&v1] () mutable {return ++v1;};
        v1 = 0;
        auto j = f(); // j has the value 1
}
```

Lambda return statements

- We do not have to specify a return type when lambda function contains a single return statement
- By default, the lambda is assumed to return void, if a lambda body contains any statement other than a return
  - When is Lambda is assuming to be returning a void it cannot return a value

Binding arguments

- Lambda functions however did not solve our original problem that the value v has to be defined in advanced in a local variable
- The problem still persist because in the lambda function we have captured the variable v, and not passing is as argument
- The library bind can solve the problem of passing a size argument to a given lambda function
- The library bind takes a callable object and generates another callable object that adapts the parameter list to the original one

Binding arguments

- The general form of a call to bind is

auto new_CaptFn = bind(callable, arg_list)

- callable is itself a callable object

- arg_list is a comma-separated list of arguments which correspond to the parameters of the given callable
  - The arguments in arg_list may include names of the form _n, where n is a integer
  - The number n is the position of the parameter in the generated callable
    - _1 is the first parameter of the new_CaptFn
    - _2 is the second parameter
    - so on

Binding arguments

```
auto f = [] (int x, int value) -> bool {return x > value;}

auto p = find_if(vec.begin(), vec.end(), bind(f,_1,56);
```

- The placeholder names are in the namespace std::placeholders
- The bind function is defined in the functional header
- By default the arguments to bind that are not placeholders are copied
  - We can use the library ref function to pass a reference

Numerical algorithms

- Most of the standard algorithms deal with the usual data management issues – such as copy, sort, search
- There are also numerical algorithms can be important when you compute within the STL framework
- We look at one an example of one here – accumulate

Accumulate

```
template<class In, class T>
  T accumulate(In first, In last, T init)
    {
        while (first!=last) {
                init = init + *first;
                 ++first;
        }
        return init;
     }
```

Accumulate

- Given an initial value, the algorithm simply adds all the elements in [first,last]

```
int a[] = {1, 2,3, 4, 5};
cout << accumulate(begin(a), end(a), 0);
```

- The type of the *init* variable will determine the *return* type of the accumulator

Accumulate

- The library offers a second version of the accumulate which allows to specify an operation which takes two arguments

```
template<class In, class T, class AccumOp>
  T accumulate(In first, In last, T init, AccumOp op)
  {
          while (first!=last) {
                  init = op(init,*first);
                   ++first;
          }
          return init;
}
```

- We can instantiate this template which any function that takes two arguments of the accumulator type and return an element of the accumulator type multiplies<T>, divides<T>, minus<T>, modulus<T> which are defined in functional any other that we can define