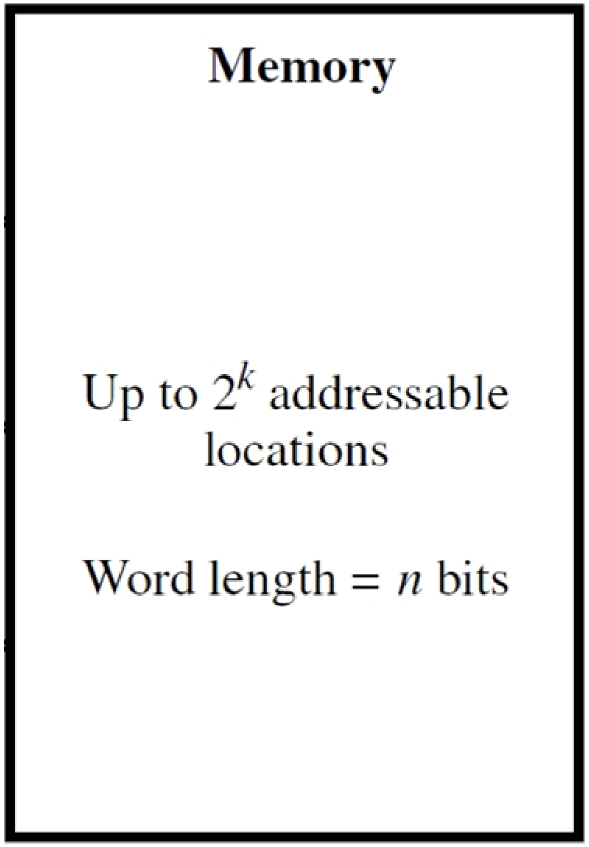
EECE 2560:
Fundamentals of Engineering Algorithms

C++ Programming Review

Memory Addresses

- Pointers are merely the addresses of where data are stored in the computer main memory.
- 2^k x *n* memory has 2^k locations with *n* bits in each location.
- Memory address:
  - unique (*k*-bit) identifier of location
- Memory contents:
  - *n*-bit value stored in location

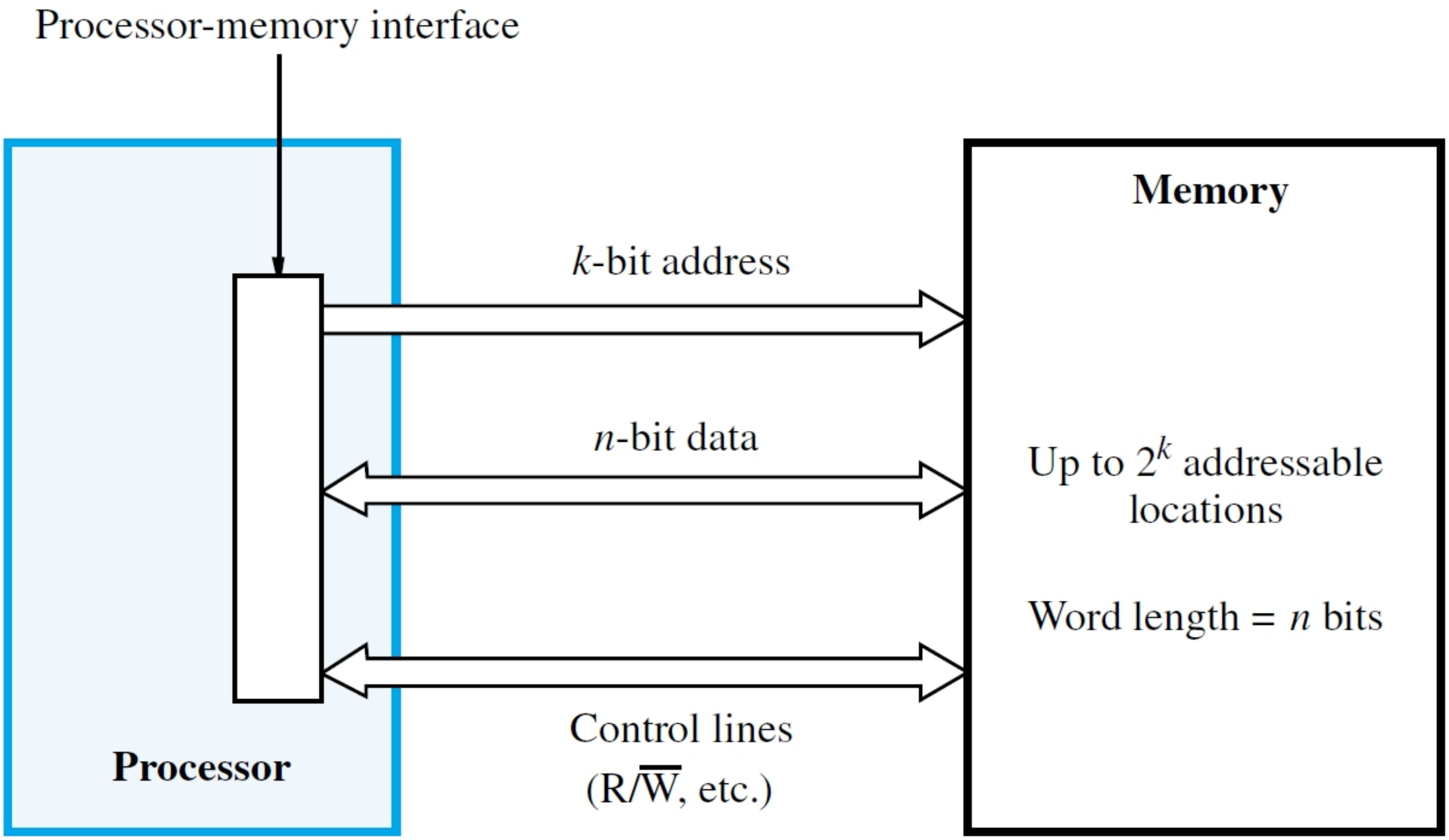


Memory

Up to 2^k addressable locations

Word length = *n* bits

Processor/Memory Interface


Processor-memory interface
Memory
k-bit address
n-bit data
Up to 2^k addressable locations
Word length = n bits
Control lines
(R/$\overline{\text{W}}$, etc.)
Processor

Pointers

- A pointer is merely an address of where a datum or structure is stored
  - Pointers operations are based on the type of entity that they point to.
  - To declare a pointer, use * preceding the variable name: int *p;
- To set a pointer to a variable's address use & before the variable as in p = &x; & means "return the memory address of"
  - in this example, p will now point to x
- If you access p, you merely get the address
- To get the value that p points to, use * as in *p
  - *p = *p + 1; will add 1 to x
- * is known as the *indirection* (or dereferencing) operator because it requires a second access
- *p and x are known as aliases, two ways of accessing the same memory location.

Example 1

```
#include <iostream>
using namespace std;
int main()
{
 int x = 1, y = 2;
 int z[3] = {10, 20, 30};
 int *p;
 p = &x;         // p now points at the location where x is stored
 y = *p;         // set y equal to the value pointed to by p, or y = x
 cout << "x=" << x <<"  y=" << y <<"\n"; // x=1  y=1
 *p = 0;         // now change the value that p points to to 0, so now x = 0
                 // but will that change y's value?
 cout << "x=" << x <<"  y=" << y <<"\n"; // x=0  y=1
 p = &z[0];      // now p points at the first location in the array z, z[0]
 *p = *p + 1;    // the value that p points to (z[0]) is incremented
 ++p;            // now p points at the second location in the array z, z[1]
 *p = *p + 1;    // the value that p points to (z[1]) is incremented
 cout << "z[0]=" << z[0] <<"  z[1]=" << z[1] <<"\n"; // z[0]=11  z[1]=21
}
```

Passing Arguments to Functions

- There are three ways in C++ to pass arguments to a function
  - pass-by-value
  - pass-by-reference
  - pass-by-pointer

Pass by Value

- When an argument is passed by value, a *copy* of the argument's value is made and passed (on the function call stack) to the called function.
  - Changes to the copy do not affect the original variable's value in the caller.
- The main disadvantage of pass-by-value is that, if a large data item being passed, copying that data can take a considerable amount of memory space and execution time.

Pass by Reference (1 of 2)

- With *pass-by-reference*, the caller gives the called function the ability to *access the caller’s data directly*, and to *modify* that data.
- A reference parameter is an *alias* for its corresponding argument in a function call.
- To indicate that a function parameter is passed by reference, simply follow the parameter’s type in the function prototype by an *ampersand* (&); use the same convention when listing the parameter’s type in the function header.
- Pass by reference is good for performance reason, because it eliminates the overhead of pass-by-value. However, it can weaken security as the called function can corrupt the caller’s data.

Pass by Reference (2 of 2)

- To specify that a reference parameter should not be allowed to modify the corresponding argument, place the `const` qualifier before the type name in the parameter’s declaration.
- Using `const` reference parameter is recommended for passing large objects to simulate the appearance and security of pass-by-value and avoid the overhead of passing a copy of the large object.

Pass by Pointer

- Another way to avoid the overhead of copying large objects to a function is to pass pointer to that object (its address) to the large data objects.
- You can use pointers and the indirection operator (*) to accomplish pass-by-pointer.
- The called function can then access the original object simply by dereferencing the pointer in the caller.

Example 2 (1 of 4)

```
//arguments.cc
//Passing arguments by value, by reference, and by pointer
#include <iostream>
using namespace std;

int squareByValue(int); // value pass
void squareByReference(int&); // reference pass
int squareByConstReference(const int&); // const reference pass
void squareByPointer(int*); // Pointer pass


int main() {
   int w{3}; // value to square using squareByValue
   int x{4}; // value to square using squareByReference
   int y{5}; // value to square using squareByConstReference
   int z{6}; // value to square using squareByPointer
```

Example 2 (2 of 4)

```
// demonstrate squareByValue
   cout << "w = " << w << " before squareByValue\n";
   cout << "Value returned by squareByValue: "
      << squareByValue(w) << endl;
   cout << "w = " << w << " after squareByValue\n\n";
// demonstrate squareByReference
   cout << "x = " << x << " before squareByReference\n";
   squareByReference(x);
   cout << "x = " << x << " after squareByReference\n\n";
// demonstrate squareByConstReference
   cout << "y = " << y << " before squareByConstReference\n";
      cout << "Value returned by squareByConstReference: "
      << squareByConstReference(y) << endl;
   cout << "y = " << y << " after squareByConstReference\n\n";
// demonstrate squareByPointer
   cout << "z = " << z << " before squareByPointer\n";
   squareByPointer(&z);
   cout << "z = " << z << " after squareByPointer\n";
}
```

Example 2 (3 of 4)

```cpp
// squareByValue multiplies number by itself, stores the
// result in number and returns the new value of number
int squareByValue(int number) {
   return number *= number; // caller's argument not modified
}
// squareByReference multiplies numberRef by itself and stores the result
// in the variable to which numberRef refers in function main
void squareByReference(int& numberRef) {
   numberRef *= numberRef; // caller's argument modified
}
// squareByConstReference returns the multiplication of numberRef by itself
int squareByConstReference(const int& numberCRef) {
   //numberCRef *= numberCRef; // Compilation error trying to modify a constant
   return numberCRef * numberCRef;
}
// squareByPointer multiplies number pointed to by numberPnt by itself
// and stores the result in the original variable
void squareByPointer(int* numberPnt) {
   *numberPnt = *numberPnt * *numberPnt; // caller's argument modified
}
```

Example 2: Sample Run (4 of 4)

```
w = 3 before squareByValue
Value returned by squareByValue: 9
w = 3 after squareByValue

x = 4 before squareByReference
x = 16 after squareByReference

y = 5 before squareByConstReference
Value returned by squareByConstReference: 25
y = 5 after squareByConstReference

z = 6 before squareByPointer
z = 36 after squareByPointer
```

Arrays

- We declare an array using [ ] following the variable name
  - `int x[5];`
- You must include the size of the array in the [ ] when declaring **unless** you are also initializing the array to its starting values as in:
  - `int x [] = {1, 2, 3, 4, 5};`
  - you can also include the size when initializing as long as the size is >= the number of items being initialized (in which case the remaining array elements are uninitialized)
  - you access array elements using indices e.g. `x[4]`
  - array indices start at 0
  - arrays can be passed as parameters, the type being received would be denoted as `int x[]`

Source: Prof. Kimani, Northeastern University

Arrays and Pointers

- In C++, the name of the array is actually a pointer to the first array element.
  `int z[5] → z = &z[0]`
- Therefore, there are two ways to access the first element of array z: either `z[0]` or `*z`
- What about accessing `z[1]`?
  - We can do `z[1]` as usual, or we can add 1 to the location pointed to by z, that is *(z+1)
- While we can update a pointer value (as we did with ++p in a previous Example), we cannot update the value of z (e.g. ++z) , otherwise we would lose access to the first array location since z is our array variable.
  - Notice that when we did p++, the value stored in p is increased by 4. This is because p is of type "`int *`" and an integer size is 4 bytes.
- If p is pointing to doubles, the increment ++p would increment p by 8 bytes away, and by 1 if it points to char type.

Source: Prof. Kimani, Northeastern University

Iterating Through the Array

- The following two ways to iterate through an array, the usual way, but also a method using pointer arithmetic:

```
int j;
int z[3] = {10, 20, 30};
 for(j = 0; j < 3; j++)
     cout << z[j] << << "\n";
```

```
int *p;
int z[3] = {10, 20, 30};
 for(p = z; p < z + 3; p++)
       cout << *p << << "\n";
```

- For the code on the right:
  - p started by having the value of z, that is the address of `z[0].`
  - The loop iterates while p `< z + 3;` where z+2 is the address of the last element of the 3-element array z.
  - `p++` increments the pointer to point at the next element in the array.
- *Note:* `(*p)++;` increments the value of the element to which p points. While `*(p++);` increments the pointer to point at the next array element and then return the value of that element.

Source: Prof. Kimani, Northeastern University

Example 3

```
#include <iostream>
using namespace std;
int main()
{
 int x[4] = {12, 20, 39, 43}, *y;
 y = x;                      // y points to the beginning of the array
 cout << x[0]  <<endl;       // outputs 12  (note: endl outputs a newline)
 cout << *y    <<endl;       // also outputs 12
 cout << *y+1  <<endl;       // outputs 13 (12 + 1)
 cout << *(y+1)<<endl;       // outputs x[1] or 20
 y+=2;                       // y now points to x[2]
 cout << *y    <<endl;       // prints out 39
 *y = 38;                    // changes x[2] to 38
 cout << *y-1  <<endl;       // prints out x[2] - 1 or 37
 y++;                        // sets y to point at the next array element
 cout << *y    <<endl;       // outputs x[3] (43)
 (*y)++;                     // sets what y points to, to be 1 greater
 cout << *y    <<endl;       // outputs the new value of x[3] (44)
}
```

Source: Prof. Kimani, Northeastern University

Passing Arrays to Functions

- When an array is passed to a function, what is being passed is a pointer to the array
  - In the formal parameter list, you can either specify the parameter as an array or a pointer

```
int array[100];
…
afunction(array);
…

void afunction(int *a) {…}
      or
void afunction(int a[]) {…}
```

Source: Prof. Ubal, Northeastern University

Example 4

- Write a program that finds the minimum value in an array of pre-initialized integers. Encapsulate the search functionality in a function that takes an array and its size as arguments.
- Solution:

```
#include <iostream>
using namespace std;
int Min(int v[], int n)
{
// Initialize minimum value to the first element
int result = v[0];
// Traverse the rest of the array
for (int i = 1; i < n; i++)
if (result > v[i])
result = v[i];
// Return minimum value
return result;
}
```

```
int main()
{
 // Declare array
 int v[] = {5, 6, 7, 8, 2, 1};
 // Obtain minimum value
 int min = Min(v, 6);
 // Print result
 cout << "The minimum value is ";
 cout << min << '\n';
 return 0;
}
```

Source: Prof. Kimani, Northeastern University

Void and Null

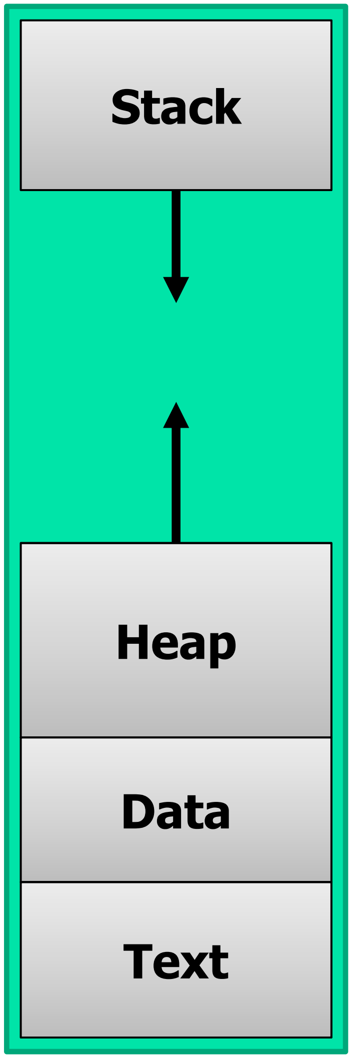
- We can declare a pointer to point to a void type, which means that the pointer can point to any type.
- However, this requires a cast before the pointer can be assigned

```
int x;
float y;
void *p;           // p can point to either x or y
p = (int *) &x;    // p can point to int x once the address is cast
p = (float *) &y;  // or p can point to float y
```

- Pointers that don't currently point to anything have the special value NULL and can be tested as (p == NULL) or (!p), and (p != NULL) or (p)

Program Memory Layout

- **Text**: program code
- **Data**: global variables (allocated upon program start)
- **Stack**: local variables
- **Heap**: Dynamic memory



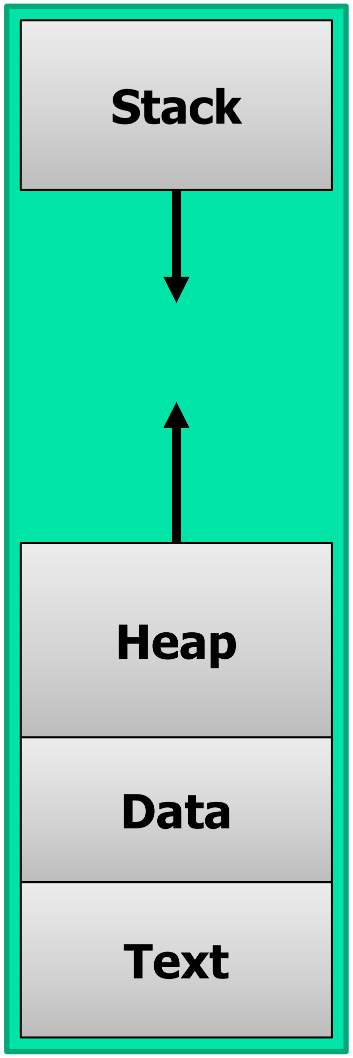
```
. . .
int ClassSize;
int main() {
  int CurrentGrades[100];
  //..... read current class size and grades
  cout << "Grades average = ";
  cout << AverageGrades(CurrentGrades) << endl;
}
double AverageGrades(int grades[]){
    double sum = 0;
    for(int i = 0; i < ClassSize; i++)
        sum += grades[i];
    return sum/ClassSize;
}
```

Source: Prof. Kimani, Northeastern University

Stack

- Memory for C/C++ run-time system to keep track of active functions
  - Stack pointer (SP)
- Upon function invocation, create "stack frame" containing
  - Return address
  - Return value
  - Local variables, …
- Upon return, pop frame from stack and continue at return address.

```
int main(void) {
   int i = 5;
   foo(i);
   return 0;
}
void foo(int j) {
   int k;
   k = j+1;
   bar(k);
}
void bar(int m) {
   /* ... */
}
```



Source: Prof. Kimani, Northeastern University

Dynamic Memory Allocation

- To this point, we have been declaring pointers and having them point at already created variables/structures.
- An important use of pointers is to create dynamic structures.
  - structures that can have data added to them or deleted from them such that the amount of memory being used is equal to the number of elements in the structure
  - this is unlike an array which is static in size.
- Creating and maintaining dynamic data structures requires dynamic memory allocation – the ability for a program to obtain more memory space at execution time and to release space no longer needed.

Source: Prof. Kimani, Northeastern University

Dynamic Memory Allocation

- In C++, the *new* operator and the *delete operator* are essential to dynamic memory allocation.
- Operator *new* is used to request memory space enough to hold a specific data type or an array of the data type.
- Dynamically allocate memory to an integer

```
int *ptr = new int;
```

- Dereference the pointer to access the memory

```
*ptr = 8;            // assings 8 to memory
```

- At the end free up the memory for reuse

```
delete ptr;          // returns memory to the system.
```

Dynamic Memory Operators for Arrays

- Dynamically allocate memory to an integer array with 10 elements

```
int *arr = new int[10];
```

- Dereference the pointer to access the memory

```
arr[0] = 8; // access element at index 0
```

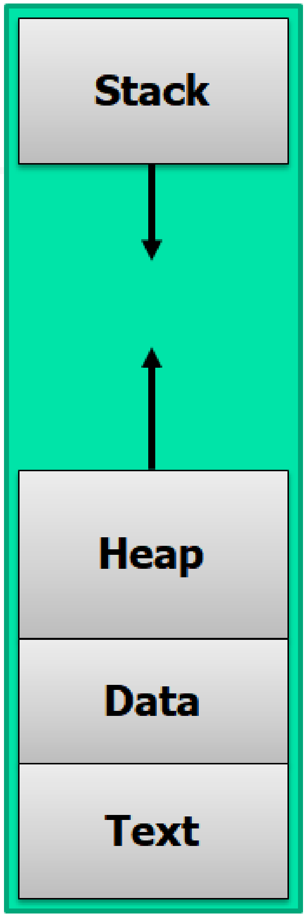
- At the end free up the memory for reuse

```
delete[] arr; // free up memory.
```

Note the use of `[]` for arrays

Example 5

```
//Program to read and calculate the average of class grades
#include <iostream>
using namespace std;
double AverageGrades(int grades[]);
int ClassSize;

int main()
{
  int* gradesP;
  cout<< "What is the class size? ";
  cin>> ClassSize;
  gradesP = new int[ClassSize];
  for(int i = 0; i < ClassSize; i++) {
   cout << "What is grade # "<<i+1<<"? ";
   cin >> gradesP[i];
  }
  cout << "Grades average = ";
  cout << AverageGrades(gradesP);
  cout << endl;
  delete[] gradesP;
}
```



```
double AverageGrades(int grades[]){
    double sum = 0;
    for(int i = 0; i < ClassSize; i++)
        sum += grades[i];
    return sum/ClassSize;
}
```

Source: Prof. Kimani, Northeastern University

Object Definition

- An object can be “a tangible or an abstract” entity that can be separated with unique properties.
  - Examples of objects: a person, a book, a car, a computer.
- An object can contain other objects:
  - a car = wheels + engine + door + windshield + ...
  - a house = kitchen + bedrooms + living room + ...
  - a laptop = keyboard + display + processor + ...
- Each object has a set of attributes (properties), such as location, speed, size, address, color ...
- Each object has its behaviors (functions/methods), such as ring (phone), accelerate and move (car), take picture (camera), ...

Source: Prof. Kimani, Northeastern University

Object Oriented Programming

- In object oriented programming (OOP), a class defines the common properties and behavior of a set of objects.
  - Example: the class `Cars` defines the properties and behavior of any car. My car and your car are two objects that represent two “instances” of that class.

- The class datatype is the cornerstone of C++ . It gives C++ a unique identity versus C.

Source: Prof. Kimani, Northeastern University

Why OOP?

- It provides more natural description of the interactions between the program components (objects).
- It provides easier and safer code reuse (e.g., class inheritance)
- It protects data through encapsulation: hide data that do not have to be visible to the other objects or protect data from unintentional changes.
- It allows class member functions with fewer or no arguments as the data is already defined in the class.

Source: Prof. Kimani, Northeastern University

Example 6

```
//rectangle.cc
#include <iostream>
using namespace std;

class Rectangle
{ private:
    int width;
    int length;
  public:
    void set(int w, int l) {
     width = 0; length = 0;
     if (w>0) width=w;
     if (l>0) length=l;     }

    int area() {
     return width * length; }
};
```

```
int main() {
 Rectangle  r1;
 Rectangle  r2;

 r1.set(5, 10);
 cout <<r1.area() << endl;
 r2.set(-1, -2);
 cout <<r2.area() << endl;
}
```

Source: Prof. Kimani, Northeastern University

Access Control

- Information hiding - encapsulation
  - To prevent the internal representation from direct access from outside the class
- Access Specifier Keywords
  - `public`: may be accessible from anywhere within a program
  - `private` (default): may be accessed only by the member functions, not open for non-member functions

Source: Prof. Kimani, Northeastern University

Example 7

- Class member functions have to be declared inside the class body. However, their definition can be placed inside the class body (as in the previous Example), or outside the class body as in this example:

```
//circle.cc
#include <iostream>
using namespace std;

class Circle {
  private:
   double radius;
  public:
   void setRadius(double);
   double getArea(void);
   void displayRadius(void);
};
// member function definitions
void Circle::setRadius(double r)
{ if (r>0) radius = r;
  else radius = 0; }
```

```
double Circle::getArea()
{   return 3.14*radius*radius; }

void Circle::displayRadius()
{ cout << "r = " << radius << endl;
}

int main() {
 Circle c; // an object of circle class
 c.setRadius(5.0);
 cout << "Circle area =";
 cout << c.getArea() << endl;
 c.displayRadius();
}
```

Source: Prof. Kimani, Northeastern University

Example 8

- In this example, we create an array of Circle objects:

```
//circlesArray.cc
#include <iostream>
using namespace std;
class Circle {
  private:
   double radius;
  public:
   void setRadius(double);
   double getArea(void);
   void displayRadius(void);
};
// member function definitions
void Circle::setRadius(double r)
{ if (r>0) radius = r;
  else radius = 0; }

double Circle::getArea()
{   return 3.14*radius*radius; }
```

```
void Circle::displayRadius()
{ cout << "r = " <<radius<< endl; }

int main() {
 Circle myCrcls[5]; //five Circle objects.
 myCrcls[0].setRadius(2.0);
 myCrcls[1].setRadius(4.0);
 myCrcls[2].setRadius(8.0);
 myCrcls[3].setRadius(16.0);
 myCrcls[4].setRadius(32.0);

 cout << "\nCircles areas are: ";
 for (int i = 0; i <= 4; i++){
   cout << "\nCircle " << i+1;
   cout <<":"<<myCrcls[i].getArea();
  }
}
```

Source: Prof. Kimani, Northeastern University

Objects Constructors

- Constructors ensure that the instances of a class (objects) are properly initialized.
- A constructors looks like a function with the same name as the class itself.
- Constructors have no return type.
- Constructors go in the `public` part of the class.
- A class may have multiple overloaded constructors (each with different parameters)
- Constructors are invoked automatically without explicit function call.

Source: Prof. Kimani, Northeastern University

Objects Destructors

- Destructors ensure that the instances of a class (objects) are properly destroyed when they are not needed any more.
- A destructor looks like a function with the same name as the class itself but preceded with the tilde symbol ~
- Destructors have no return type
- Destructors go in the `public` part of the class
- Destructors do not take parameters and hence there is only one destructor per class.
- Invoked automatically without explicit function call

Example 9 (1 of 3)

```
//ConstructorsDestructors.cc
#include <iostream>
using namespace std;

class Person {
  private:
   char * name;
   int    age;
  public:
   Person(); //constructor with no arguments (default constructor)
   Person(int nsize); //constructor with one arguments for name size
   Person(int nsize, int agev); //const. with 2 arguments for name size and age
   void setAge(int agev) {age = agev>0?agev:20;}
   void getName() {cin >> name;}
   void PrintInfo();
   ~Person() //destructor
      { delete [] name;    } //free the memory allocated for name
};
```

Example 9 (2 of 3)

```
// member function definitions
Person::Person() {
   name = new char[100]; //default name size is 100 characters
   age = 20; //default age 20
}

Person::Person(int nsize) {
   name = new char[nsize];
   age = 20; //default age     }

Person::Person(int nsize, int agev){
   name = new char[nsize];
   setAge(agev);                    }

void Person::PrintInfo() {
   cout << "Name: " << name;
   cout << " Age: " << age <<endl; }
```

Example 9 (3 of 3)

```
int main() {
 Person P1, P2(50); //or P2{50}. Bracket are required if this object is declared as
                    //a member of another class as parenthesis would confuse it
                    //with a definition of a method of that class.
 Person *P3p = new Person(30, 21);

 cout << "P1's name? "; P1.getName();
 P1.setAge(-22);
 cout << "P2's name? "; P2.getName();
 P2.setAge(25);
 cout << "P3's name? "; P3p->getName();

 cout << "Person 1 "; P1.PrintInfo();
 cout << "Person 2 "; P2.PrintInfo();
 cout << "Person 3 "; P3p->PrintInfo();
 delete P3p;
}
```

Source: Prof. Kimani, Northeastern University

Dynamic Memory Allocation

- In the previous Example, constructors dynamically allocate memory for the `name` member.
  - The destructor deletes this allocated memory and is automatically invoked once the object reaches end of its lifetime.
- In the same example, the object pointed to by `P3p`, is created by calling “`new`”. Here the whole object has memory dynamically allocated for it.
  - This dynamically created object has to be deleted by calling “`delete`”, which automatically calls the object’s destructor.
- *Note:* Do not use `malloc` / `free` in C++. They do not call the constructor / destructor.

Memory Leak

- In C++, it is very important to reclaim any allocated memory otherwise the program will gradually run out of memory and eventually crash.
- Memory leak is hard to detect by running the program
- Leaks usually takes care in a small amount each time
- It might take hours, days, or weeks to run out of memory
- Memory leak sometime is mistaken as performance problems.

Operator Overloading

- Overloaded operators provide a concise notation for manipulating string objects.
- You can use operators with your own user-defined classes.
- Although C++ does not allow new operators to be created, it does allow most existing operators to be overloaded.
- Operator overloading is not automatic—you must write operator-overloading functions to perform the desired operations.
- An operator is overloaded by writing a non-`static` member function definition as you normally would, except that the function name starts with the keyword `operator` followed by the symbol for the operator being overloaded.
  - For example, the function name operator+ would be used to overload the addition operator (+).

Operator Overloading Restrictions

- An operator’s precedence cannot be changed by overloading.
  - However, parentheses can be used to force the order of evaluation of overloaded operators in an expression.
- An operator’s “arity” (that is, the number of operands an operator takes) cannot be change
  - overloaded unary operators remain unary operators; overloaded binary operators remain binary operators. Operators &, *, + and - all have both unary and binary versions; these unary and binary versions can be separately overloaded.
- You cannot overload operators for fundamental types
  - For example, you cannot make the + operator subtract two integers. Operator overloading works only with objects of user-defined types or with a mixture of an object of a user-defined type and an object of a fundamental type.

Example 10 (1 of 2)

```
//circle-operators.cc
//The circle class with operator overloading of == and !=
#include <iostream>
using namespace std;

class Circle {
  private:
   double radius;
  public:
   Circle(double r);  //constructor with a radius
   void setRadius(double);
   double getArea(void);
   void displayRadius(void);
   bool operator==(const Circle &c2);
   bool operator!=(const Circle &c2);
};
```

Example 10 (2 of 2)

```
// member function definitions
Circle::Circle(double r) { setRadius(r); }

bool Circle::operator==(const Circle &c2) {
       return (radius == c2.radius);     }

bool Circle::operator!=(const Circle &c2) {
       return !(*this == c2);            } // "this" will be explained later.

void Circle::setRadius(double r) {
  if (r>0) radius = r; else radius = 0; }

double Circle::getArea() { return 3.14*radius*radius; }

int main() {
 Circle cir1(5), cir2(7); // an object of circle class
 if (cir1 == cir2)  cout << "Both circuits are the same.\n";
 if (cir1 != cir2)  cout << "Both circuits are not the same.\n";
}
```

The this Pointer

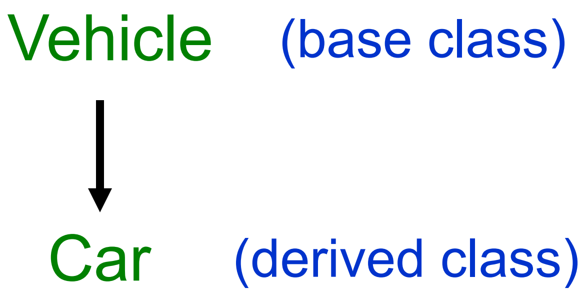
- Every object has access to its own address through a pointer called this (a C++ keyword).
- This pointer is passed (by the compiler) as an *implicit* argument to each of the object’s non-static member functions.
- Member functions use the this pointer to reference an object’s data members, for example, to avoid *naming conflicts* between a class’s data members and member-function parameters (or other local variables).

Source: Prof. Kimani, Northeastern University

Introduction to Inheritance

- The mechanism by which one class can acquires the properties (data and operations) of another class, and then extend that class
- We will utilize the “IS_A” relationship to define inheritance
  - e.g. A Car is a Vehicle
- Base Class (or superclass or parent): the class being inherited from.
- Derived Class (or subclass or child): the class that inherits

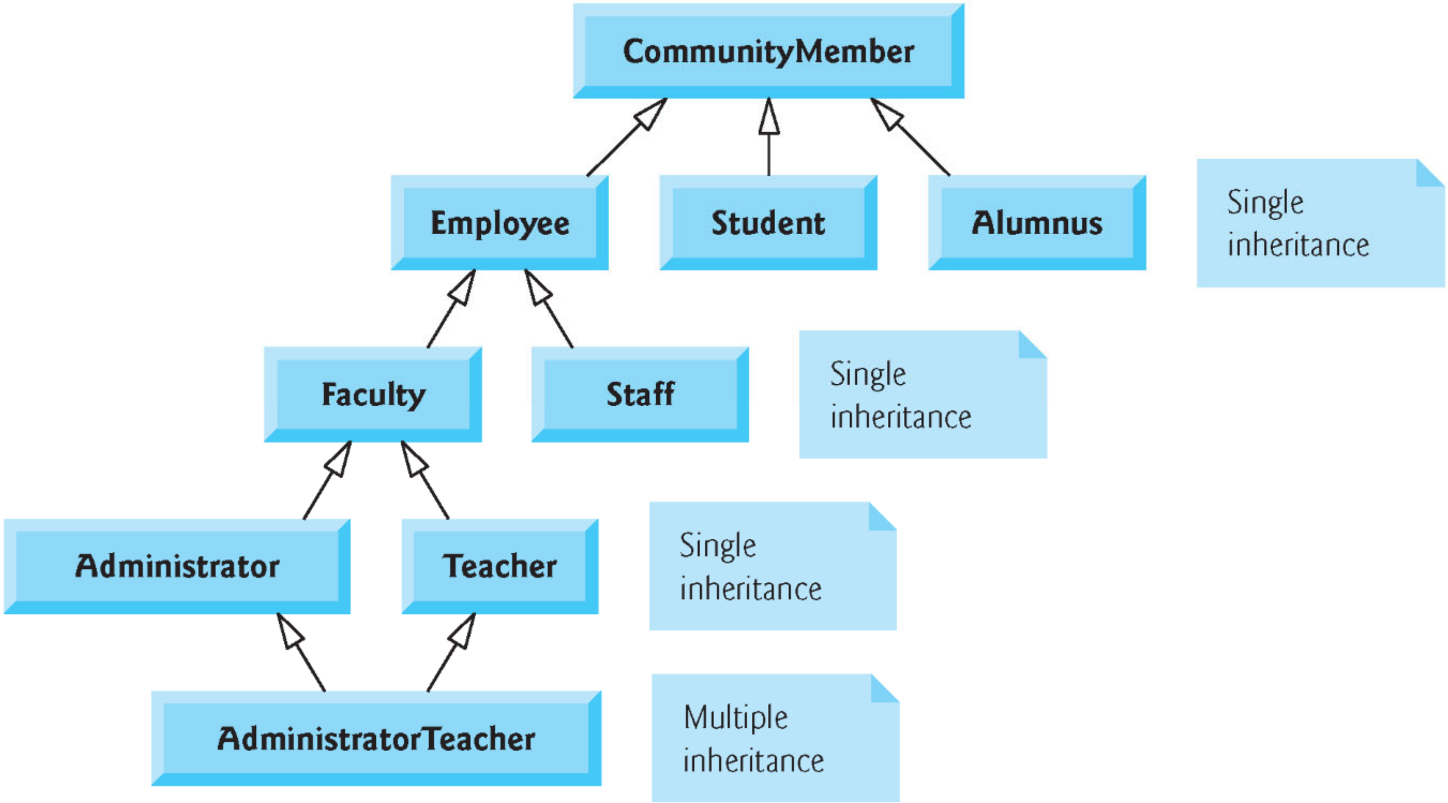
```
class Car : public Vehicle {
     private:
        ...

     public:
        ...
};
```



Source: Prof. Kimani, Northeastern University

Inheritance Hierarchy

- Concepts at higher levels are more general
- Concepts at lower levels are more specific (inherit properties of concepts at higher levels)



Source: Prof. Kimani, Northeastern University

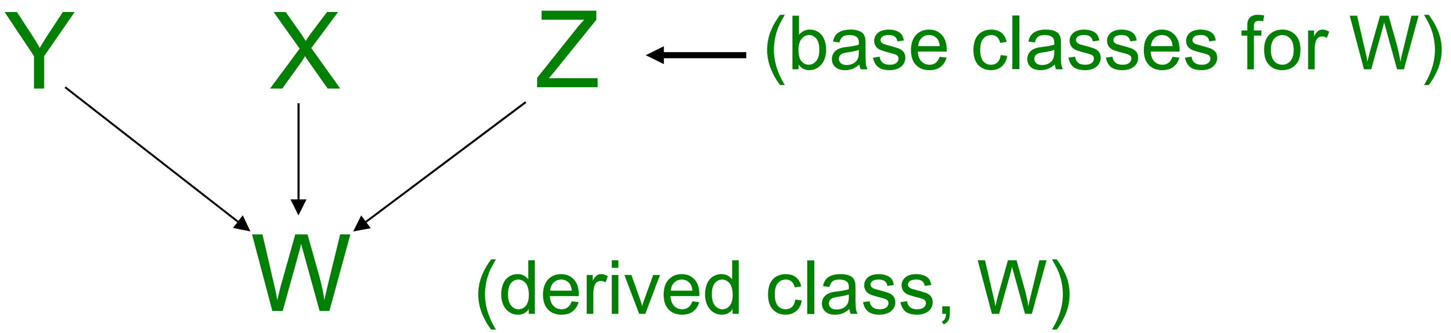
Advantages of Inheritance

- When a class inherits from another class, there are 3 benefits:

1. You can reuse the methods and data of the existing class
2. You can extend the existing class by adding new data and new methods
3. You can modify the existing class by overloading its methods with your own implementations

Source: Prof. Kimani, Northeastern University

Multiple Inheritance Example



Y X Z ← (base classes for W)

W (derived class, W)

```
class W : public Y, public X, public Z {
     private:
          ...

     public:
          ...
};
```

Constructors and Destructors

- Base class’s default constructor, is the user-defined constructor that has no arguments or it is the default constructor provided implicitly by the compiler.
  - *Note:* the compiler implicitly provides a default constructor with no parameters **only** in a class that does not explicitly define any constructors.
  - Base class’s default constructor, if provided (by the user or the compiler), is invoked automatically from derived classes in addition to their own constructors (if any defined).
  - Base class’s default constructor cannot be overridden in a derived class.
- Destructors are called in reverse order from the constructors for derived class objects.

Source: Prof. Kimani, Northeastern University

Example 11: Constructor with no Arguments (1 of 2)

```
//shape.cc
#include <iostream>
#include <string>
using namespace std;

class Shape {
private:
       int area;

public:
  Shape(){
    area = 99;
    cout << "Shape Created\n";}

  ~Shape(){
    cout<<"Shape Destroyed\n";}
};
```

```
class Circle : public Shape {
private:
       int radius;
public:
 Circle(){
   radius = 5;
   cout << "Circle Created\n";}

 ~Circle(){
   cout<<"Circle Destroyed\n";}
};

int main() {
  //create objects
  Shape myShape;
  Circle myCircle;
  cout<< "The Main code...\n";
}
```

Source: Prof. Kimani, Northeastern University

Example 11: Results (2 of 2)

```
$ g++ Example2-Shape.cc -o ex2
$ ./ex2
Shape Created
Shape Created
Circle Created
The Main code...
Circle Destroyed
Shape Destroyed
Shape Destroyed
$
```

Constructors with Arguments

- In case a base class has only constructors that require arguments, then it does not have a default constructor to be called automatically by its derived classes.
- In this case, a derived-class’s constructor is required and it has to explicitly call one of its base-class’s constructor.

Example 12: Constructor with Arguments

```
//argShape.cc
#include <iostream>
#include <string>
using namespace std;

class Shape {
private:
    int area;

public:
  Shape(int a){
    area = a;
    cout << "Shape Created\n";}

  ~Shape(){
    cout<<"Shape Destroyed\n";}
};
```

```
class Circle : public Shape {
private:
        int radius;
public:
 Circle():Shape(55){//error if Shape(55) is not called
   radius = 5;
   cout << "Circle Created\n";}

 ~Circle(){
   cout<<"Circle Destroyed\n";}
};

int main() {
  //create objects
  Shape myShape(44); //error if no parameter used
  Circle myCircle;
  cout<< "The Main code...\n";
}
```

Example 13 (1 of 4)

```
//employee-manager.cc
#include <iostream>
using namespace std;

class Employee {
protected:
       float monthlySalary;

public:
  Employee() {monthlySalary = 1000;} //1st Constructor
  Employee(float salary) {monthlySalary = salary;} //2nd Constructor
  float getSalary() {return monthlySalary;}
  float getMonthlyPayment() { return monthlySalary; }
};
```

Example 13 (2 of 4)

```
class Manager : public Employee {
protected:
  float monthlyBonus;

public:
  Manager(){monthlyBonus = 100; } //1st Constructor
  Manager(float salary, float bonus): Employee(salary) //2nd Constructor
                                              //If Employee(salary) is omitted,
                                              //the default Employee() would be
                                              //called (setting the salary to 1000
     {monthlyBonus = bonus;}

  float getMonthlyBonus() {return monthlyBonus;}

  float getMonthlyPayment() {return monthlySalary+monthlyBonus;}
};
```

Example 13 (3 of 4)

```
int main() {
 Employee E1;
 Manager M1;
 Employee E2(3000.0);
 Manager M2(5000.0, 500.0);

  cout<<" First employee's Salary = "<< E1.getSalary();
  //cout<<"  His monthly bonus"<< E1.getMonthlyBonus();  // Not allowed! No access to derive classes
  cout<<"\n\t His monthly payment = "<< E1.getMonthlyPayment();

  cout<<"\n First manager's Salary = "<< M1.getSalary();
  cout<<"\n\t Her monthly bonus = "<< M1.getMonthlyBonus();
  cout<<"\n\t Her monthly payment = "<< M1.getMonthlyPayment();

  cout<<"\n Second employee's Salary = "<< E2.getSalary();
  cout<<"\n\t Her monthly payment = "<< E2.getMonthlyPayment();

  cout<<"\n Second manager's Salary = "<< M2.getSalary();
  cout<<"\n\t His monthly bonus = "<< M2.getMonthlyBonus();
  cout<<"\n\t His monthly payment = "<< M2.getMonthlyPayment();
  cout << endl;
}
```

Example 13: Results (4 of 4)

```
First employee's Salary = 1000
        His monthly payment = 1000
First manager's Salary = 1000
        Her monthly bonus = 100
        Her monthly payment = 1100
Second employee's Salary = 3000
        Her monthly payment = 3000
Second manager's Salary = 5000
        His monthly bonus = 500
        His monthly payment = 5500
```

Polymorphism

- In the previous Example, the `Manager` class has the following:
  - `getSalary()` method inherited from the `Employee` class.
  - `getMonthlyPayment()` method has a new version that overrides the original method in `Employee`.
  - `getMonthlyBonus()` method is added.
- C++ polymorphism allows a call to a member function to execute a different function depending on the type of object that invokes the function.
- It also allows a pointer to a child class to be safely converted into a pointer to any of its parent classes, but not necessarily vice versa.
- The following example shows which version of the overridden methods would be called by such pointers.

Example 14 (1 of 2)

- The following code uses the `Employee` and `Manager` classes defined in the previous Example.

```
//poly-employee.cc
int main() {
 Employee E3(3000.0);//create Employee object
 Manager M3(5000.0, 500.0); //create Manager object
 Employee *EP1, *EP2; //pointers to Employee objects

 EP1 = &E3;
  cout<<" Employee's Salary = "<< EP1 -> getSalary();
  cout<<"\n\t  his monthly payment = "<< EP1 -> getMonthlyPayment();

 EP2 = &M3;
  cout<<"\n Manager's Salary = "<< EP2 -> getSalary();
  //cout<<"  her monthly bonus = "<< EP2 -> getMonthlyBonus();
                          //Error. Method is not member of Employee
  cout<<"\n\t her monthly payment = "<< EP2 -> getMonthlyPayment();
  cout << endl;
}
```

Example 14: Results (2 of 2)

```
Employee's Salary = 3000
         his monthly payment = 3000
Manager's Salary = 5000
        her monthly payment = 5000
```

- Both the Employee and Manager objects called the employee's getMonthlyPayment() function because pointers to Employee are used to invoke the function for both objects.

Virtual Methods

- The previous example shows that it is the type of the pointer (i.e., `Employee`), not the type of the object it points to (i.e., `Manager`) that determines which methods, and which version of overridden methods, will be called.
  - That's why pointer EP2 cannot call `getMonthlyBonus()` and it calls the `getMonthlyPayment()` version of the `Employee` class not the `Manager` class.
- What if we prefer that it calls the version of `getMonthlyPayment()` that corresponds to the type of the object pointed to (i.e., `Manager`).
- We can get this behavior by making the overridden method virtual as in the following example.
  - A virtual method is a method in a base class whose behavior can be overridden by a function in a derived class with the same name, arguments, and return value.

Example 15 (1 of 3)

```
//vir-employee.cc
class Employee {
protected:
  float monthlySalary;
public:
  Employee() {monthlySalary = 1000;} //1st Constructor
  Employee(float salary) {monthlySalary = salary;} //2nd Constructor
  float getSalary() {return monthlySalary;}
  virtual float getMonthlyPayment() { return monthlySalary; }
};
class Manager : public Employee {
private:
  float monthlyBonus;
public:
  Manager() {monthlyBonus = 100; } //1st Constructor
  Manager(float salary, float bonus): Employee(salary) //2nd Constructor
     {monthlyBonus = bonus;}
  float getMonthlyBonus() {return monthlyBonus;}
  virtual float getMonthlyPayment()
        {return   monthlySalary+monthlyBonus;}
};
```

Example 15 (2 of 3)

```
int main() {
 Employee E4(3000.0);//create Employee object
 Manager M4(5000.0, 500.0); //create Manager object
 Employee *EP1, *EP2; //pointers to Employee objects

 EP1 = &E4;
  cout<<" Employee's Salary = "<< EP1 -> getSalary();
  cout<<"\n\t his monthly payment = "<< EP1 -> getMonthlyPayment();
  cout << endl;

 EP2 = &M4;
  cout<<" Manager's Salary = "<< EP2 -> getSalary();
 //cout<<" his monthly bonus = "<< EP2 -> getMonthlyBonus();
                              //Error. Method is not member of Employee
  cout<<"\n\t his monthly payment = "<< EP2 -> getMonthlyPayment();
  cout << endl;
}
```

Example 15: Results (3 of 3)

```
Employee's Salary = 3000
        his monthly payment = 3000
Manager's Salary = 5000
        his monthly payment = 5500
```

- Each one of the `Employee` and `Manager` objects called its own `getMonthlyPayment()` function even though pointers to `Employee` are used to invoke the function for both objects.

Pure Virtual Methods

- A pure virtual method is a virtual method with no implementation in the base class where it is declared, forcing all derived classes to override its behavior.
- This is how you declare a pure virtual method:

```
    virtual float getMonthlyPayment() = 0;
```

- A class containing at least one pure virtual function is called an ***abstract*** class. Abstract classes cannot be instantiated.

Extending a Base Class Method

- In addition to inheriting, adding new, overriding methods; a derived class can extend the task of a base class method as shown in the following example:

```
class Parent {
private:
    int x;
public:
    Parent() { x = 5;}
    void printInfo() { cout << " x = " << x << endl; }
};

class Child : public Parent {
private:
    int y;
public:
    Child() { y = 9; }
    void printInfo() {
            Parent::printInfo();
            cout << " y = " << y << endl; }
};
```

Public and Private Inheritance

```
class X : private Y
{         ... };
```

```
class X : public Y
{         ...   };
```

- The default inheritance is `private`.
- With private inheritance, public members of Y become private in X.
- With public inheritance, public members of Y retain their status in X.

Example 16: Private Inheritance

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

class Base {
private:
      int bPrivate;
public:
       int bPublic;
};
```

Note: no need to write “`private Base`” as it is the default inheritance type.

```
class Derived : private Base {
public:
     Derived() {
     bPrivate = 10; //Not ok
     bPublic  = 20; //OK
    }
};

int main() {
  Base cBase;
  cBase.bPublic = 30; //OK
  cBase.bPrivate = 40; //Not ok

  Derived drv;
  drv.bPublic = 50; //Not ok
  drv.bPrivate = 60;  //Not ok
}
```

Example 17: Public Inheritance

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

class Base {
private:
      int bPrivate;
public:
       int bPublic;
};
```

Note: here you need to write “`public Base`”.

```
class Derived : public Base {
public:
     Derived() {
     bPrivate = 10; //Not ok
     bPublic  = 20; //OK
   }
};

int main() {
  Base cBase;
  cBase.bPublic = 30; //OK
  cBase.bPrivate = 40; //Not ok

  Derived drv;
  drv.bPublic = 50; //ok
  drv.bPrivate = 60;  //Not ok
}
```

Function Templates (1 of 2)

- Function templates come convenient if the program logic and operations are *identical* for different data types.
- You write a single function template definition.
- Given the argument types provided in calls to this function, C++ automatically generates separate function template specializations to handle each type of call appropriately.

Function Templates (2 of 2)

- In C++, all function template definitions begin with the template keyword followed by a template parameter list enclosed in angle brackets (< and >).
- Every parameter in the template parameter list is preceded by keyword `class` or keyword `typename`.
- The type parameters are placeholders for fundamental types or user-defined types.
  - Used to specify the types of the function’s parameters, to specify the function’s return type, and to declare variables within the body of the function definition.

```
template <class T , class U> // Template parameters declaration
                   // T and U can be replaced by any identifiers of your choice.
T myFunction(T a, U b) {
      T c;
      <your code that updates c based on a and b values>
      return c;
}
```

Example 18: Header File (1 of 4)

- Assume a `maximum` function that determines the largest of three values. The following is the header file that contains the function template.

```
// maximum.h
template <class T>   // or template<typename T>
T maximum(T value1, T value2, T value3) {
    T maximumValue = value1; // assume value1 is maximum

    // determine whether value2 is greater than maximumValue
    if (value2 > maximumValue) {maximumValue = value2;}

  // determine whether value3 is greater than maximumValue
    if (value3 > maximumValue) {maximumValue = value3;}
    return maximumValue;
}
```

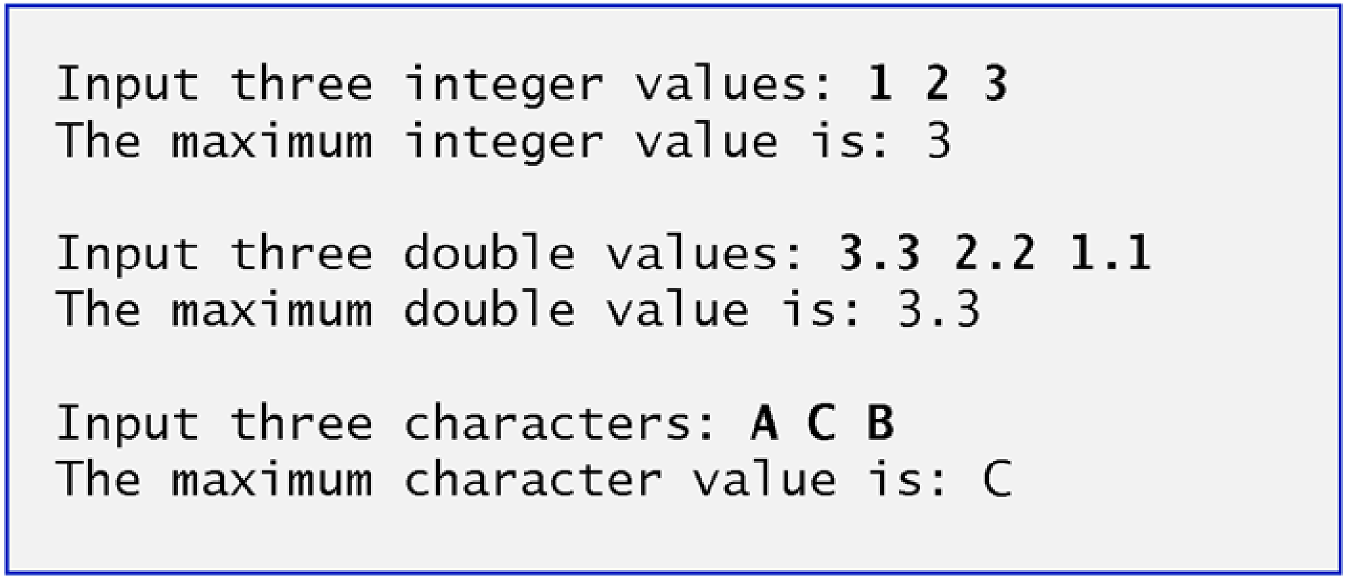
Example 18: Test Program (2 of 4)

```
// ex1-max.cc
#include <iostream>
#include "maximum.h"  // include definition of function template maximum
using namespace std;

int main() {
  // demonstrate maximum with int values
    cout << "Input three integer values: ";
    int i1, i2, i3;
    cin >> i1 >> i2 >> i3;

  // invoke int version of maximum
    cout << "The maximum integer value is: "
      << maximum(i1, i2, i3);
```

Example 18: Test Program (3 of 4)

```
// demonstrate maximum with double values
   cout << "\n\n Input three double values: ";
   double d1, d2, d3;
   cin >> d1 >> d2 >> d3;
 // invoke double version of maximum
   cout << " The maximum double value is: "
      << maximum(d1, d2, d3);

 // demonstrate maximum with char values
   cout << "\n\n Input three characters: ";
   char c1, c2, c3;
   cin >> c1 >> c2 >> c3;
 // invoke char version of maximum
   cout << "  The maximum character value is: "
      << maximum(c1, c2, c3) << endl;
} // end main
```

Example 18: Sample Run (4 of 4)



```
Input three integer values: 1 2 3
The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

Input three characters: A C B
The maximum character value is: C
```

Inappropriate Type For Templates

- You can use a template function with any type for which the code in the function definition is applicable.
- If the used type does not overload any of the operators used in the function, compilation error occur.
- For example, you cannot use the `maximum` template with the type parameter replaced by a type for which the assignment or comparison operators do not work at all, or do not work "correctly."

Class Templates

- Class templates enable you to conveniently specify a variety of related classes. It provides nice opportunity for software reusability.
- For example, assume you want to implement a class representing a list of items to be sorted *independent of the type of the items* being placed in the list.
- To *instantiate* an object of this list class, a data type must be specified. At that point, the compiler generates a copy of the class template in which all occurrences of the type parameter are replaced with the specified type.
- The following example defines a list *generically* then use *type-specific* versions of this generic list class.

Example 19: List Class Template (1 of 3)

```
// anyList.h
#include <iostream>
using namespace std;

template <class T> // or template <typenameT> and T can be any valid identifier
class anyList {
 private:
   T * ListElements;
   int LSize;
 public:
   anyList() { // Constructor
      ListElements = new T[100]; //default size is 100
      LSize = 100; fillList();  }

   anyList(int sz) { // Another constructor
      ListElements = new T[sz]; //user size
      LSize = sz;  fillList();  }

   ~anyList() { //Destructor
      delete [] ListElements; }
```

Example 19: Class Template (2 of 3)

```
  void fillList() {     // fill the list with elements
      cout << "Enter " << LSize << " elements: \n";
      for (int i=0; i<LSize; i++)
         cin >> ListElements[i]; }

 void dispList() {     // display the list contents
      cout << "The list elements are:\n";
      for (int i=0; i<LSize; i++)
       cout << "--> " << ListElements[i] << endl; }

 void sortList(){ //Sort List in ascending order
   int j;
   T temp;
   for (int i = 0; i < LSize; i++){ //Insertion Sort algorithm
     j = i;
     while (j > 0 && ListElements[j] < ListElements[j-1]){
       temp = ListElements[j];
       ListElements[j] = ListElements[j-1];
       ListElements[j-1] = temp;
       j--;      }  }  }
};
```

Example 19: Test Program (3 of 3)

```
//anylist-main.cc
#include <iostream>
#include <string>
#include “anyList.h"  // list class template definition
using namespace std;

int main() {
   cout << "List of integers: \n";
   anyList<int> intList(5); // create a list of integers
   intList.dispList(); intList.sortList(); intList.dispList();

   cout << "List of floats: \n";
   anyList<float> fltList(4); // create a list of float
   fltList.dispList(); fltList.sortList(); fltList.dispList();

   cout << "List of strings: \n";
   anyList<string> strList(3); // create a list of strings
   strList.dispList(); strList.sortList(); strList.dispList();
}
```

Source: Prof. Kimani, Northeastern University

Templates Summary

- Type-independent patterns that can work with multiple data types.
  - Generic programming
  - Code reusable
- Function Templates
  - These define logic behind the algorithms that work for multiple data types.
- Class Templates
  - These define generic class patterns into which specific data types can be plugged in to produce new classes.

Header Files

- The compiler knows what `int` is it’s a fundamental type that’s “built into” C++.
- The compiler does not know in advance user defined types, such as classes.
- When packaged properly, new classes can be reused by other programmers.
- It’s customary to place a reusable class definition in a file known as a header with a `.h` filename extension.
- You include (via `#include`) that header wherever you need to use the class.

User vs. Standard Headers

- In an `#include` directive, a header that you define in your program is placed in double quotes (" "), rather than the angle brackets (< >) used for C++ Standard Library headers like `<iostream>`.
- In this example, the double quotes tell the compiler that header is in the same folder as the current source code file, rather than with the C++ Standard Library headers.
- Files ending with the `.cpp` or `.cc` filename extension are source-code files.
- These define a program’s `main` function and other functions