As an IRS approved 501(c)3 non-profit organization, Boston High School Education (BHSE) is dedicated to building a cross-culture platform to help Chinese-originated students (both those in the United States and those in China) to obtain the holistic, high quality education that best maximize their unique individual potentials.

BHSE is a fast growing organization with currently more than seven thousand active members participating in its weekly seminars and discussions. Based on its strong footing in Boston, it is expanding rapidly into China and the rest of the world. BHSE is seeking a webpage development team to augment its expansion and to better serve its global members. It is expected to work independently to design, build and maintain BHSE’s website. The team should have basic database skills and ideally iOS and Android app development skills.

 BHSE will provide the team with
1. an excellent entrepreneurial opportunity to work with some fantastic people to obtain valuable real world experiences in the red-hot global e-learning industry and to contribute to your community by helping people,
2. a strong recommendation letter and the critical technical skills that support your future career endeavors, and
3. a potential full-time job pending the organization’s future funding and growth.

Sometimes you may want to define for your object a set of states or actions. For example, you could define the following states for a Student Admonisher Robot:
• observeStudent
• shoutAtStudent
• followStudent
• rechargeBattery

Using the const qualifier, you could define the following constants:
const int observeStudent = 1;
const int shoutAtStudent = 2;
const int followStudent = 3;
const int rechargeBattery = 4;

A function SetRobotState could then be defined as follows:
bool SetRobotState(int newState)
{
    ... 
    int currentState = newState;
    return executionSuccessful;
}
Enumeration Types

Enumeration types can be defined as follows:
```cpp
enum robotState {observeStudent = 1,
                 shoutAtStudent, followStudent, rechargeBattery};
```

This way we defined a new type `robotState` that can only assume four different values. These values still correspond to integers. For example,
```cpp
cout << followStudent;
```
gives you the output '3'.

The vector Container Type

The `vector` class is an alternative to the standard array type in C++. Its implementation uses the C++ template facility. This facility allows programmers to build generic (polymorphic) classes, that is, classes that are defined independently of the type of data they hold and operate on.

When you define such a class, you can use placeholders (parameters) instead of explicit type specifiers. You can then instantiate objects of that class for any compatible data type.

To use a vector, we have to include the following header file:
```cpp
#include <vector>
```
We can use vectors in two quite different ways:
- the array idiom
- the STL idiom (STL = Standard Template Library)

The array idiom is very similar to the built-in array type. We specify the type and size of the array and then manipulate elements of the array in the usual way:
```cpp
void main()
{
    int vecSize = 8;
    vector<int> intVec(vecSize);
    intVec[0] = 1;
    intVec[1] = 1;
    for (int i = 2; i < intVec.size(); i++)
        intVec[i] = intVec[i - 1] + intVec[i - 2];
    cout << intVec[intVec.size() - 1];
}
```
Output: '21' (Fibonacci)

When a vector is created, its elements are initialized with the default value for their type. However, we can specify a different value.

Example:
```cpp
vector<char> charVec(10, 'x');
```
initializes all ten elements of `charVec` with the value 'x'.
(By the way, the default value for characters is the blank character or space.)
The vector Container Type

A vector can also be initialized by specifying the first and one past the last element of an existing array.

**Example:**

```c++
int intArray[8] = {10, 20, 30, 40, 50, 60, 70, 80};
vector<int> intVec(intArray + 1, intArray + 5);
```

creates a vector of size four with elements 20, 30, 40, and 50.

A vector can also be initialized with or assigned to another vector.

---

The vector Container Type

The STL idiom is quite different from the array idiom. We start with defining an empty vector, for example:

```c++
vector<double> dblVec;
```

Then we can add items to the back of the vector (and thereby expanding the vector) using the `push_back` operation:

```c++
dblVec.push_back(3.7);
dblVec.push_back(2.2);
dblVec.push_back(-5.1);
```

results in vector of size 3 with elements 3.7, 2.2, -5.1.

---

The pair Type

Another class in the standard library that uses the template facility is the **pair class**.

It allows us to **combine two values** of either the same or different types **within a single object**.

If we want to use pairs, we must include the following header file:

```c++
#include <utility>
```

Then we can start defining pairs such as:

```c++
pair<string, double> FranksData("Frank Miller", 4.0);
```

---

The pair Type

We use **member access notation** to access the individual members of a pair:

```c++
string name = FranksData.first;
double GPA = FranksData.second;
```

We can manipulate the values of the members analogously:

```c++
FranksData.second = 1.0;
```

---

Typedef Names

For pair objects, it is often helpful to use the typedef mechanism.

This mechanism allows us to define mnemonic synonyms for built-in and user-defined data types.

**Example:**

```c++
typedef pair<string, double> studentGPA;
studentGPA frank("Frank Miller", 4.0);
studentGPA doris("Doris Carpenter", 3.5);
studentGPA jeff("Jeff Bridges", 2.47);
```

---

Typedef Names

- Typedef names can improve the readability of our programs.
- Notice that typedef names do not introduce new types but only synonyms for existing data types.

**For example, the following code is correct:**

```c++
typedef double GPA;
GPA JohnsGPA = 3.4;
double BobsGPA;
BobsGPA = JohnsGPA;
```
Class Types

In C++, classes are defined as follows:

```cpp
class Classname
{
public:
    // public member variables and member functions
    ...
private:
    // private member variables and member functions
    ...
protected:
    // protected member variables and member functions
    ...
};
```

September 27, 2016  CS410 – Software Engineering
Lecture #8: C++ Basics II

Class Types

Public members are accessible from anywhere within the program. To achieve information hiding, you should generally limit the public members of a class to a small set of member functions that allow the manipulation of the class objects.

Private members can be accessed only by the member functions and friends of its class. For information hiding, you should declare all member variables (data members) as private.

Protected members behave as public members to a derived class and as private members to the rest of the program.

Class Types

The declaration of class data members is identical to the declaration of variables.

Member functions are usually declared within the class definition and defined outside of it.

Notice that when you define member functions outside the class definition, you have to provide the class name and the scope resolution operator (::).

```cpp
class StudentAdmonisher
{
public:
    bool SetRobotState(robotState rs) { currentState = rs; return true; };
    robotState GetRobotState()           { return currentState; }
private:
    robotState currentState;
};
```

Class Types

However, short member functions can also be placed inside the class body.

**Example:**

```cpp
class StudentAdmonisher
{
public:
    bool SetRobotState(robotState rs) { currentState = rs; return true; }
    robotState GetRobotState() { return currentState; }
private:
    robotState currentState;
};
```

Class Types

Functions

- Function definition and declaration
- Default arguments
- Functions as arguments
- Overloading functions
- Inlining
- Storage classes
Function Definition and Declaration

The C++ code that describes what a function does is called the function definition. Its form is:

```
    type FunctionName(parameter-declaration-list)
    {
        statements
    }
```

Function Definition and Declaration

A function can be declared before it is defined. It can be defined later in the file or can come from a library or a user-specified file. Such a declaration is called a function prototype and has the following general form:

```
    type FunctionName(argument-declaration-list);
```

Default Arguments

- A formal parameter can be given a default argument.
- Usually, a default argument is a constant that occurs frequently when the function is called.
- Using a default argument saves writing this default value at each invocation.

Example:

```
int SqrOrPower(int n, int k = 2)   // k = 2 is default.
{
    assert(k > 1); // This function only works for
                 // exponents > 1.
    if (k == 2)
        return (n * n);
    else
        return (SqrOrPower(n, k – 1) * n);
}
```

Functions as Arguments

- Functions in C++ can be thought of as the addresses of the compiled code residing in memory.
- Functions are therefore a form of pointer.
- Functions can be passed as a pointer-value argument into another function.
Functions as Arguments

```cpp
doctor(F(double x))
{
    return (x*x + 1.0/x);
}
void plot(double Fcn(double), double x0, double incr, int n)
{
    for (int i = 0; i < n; i++)
    {
        cout << "x: " << x0 << " f(x): " << Fcn(x0) << endl;
        x0 += incr;
    }
}
int main()
{
    plot(F, 0.01, 0.01, 100);
    return 0;
}
```

Overloading Functions

- Overloading refers to using the same name for multiple meanings of an operator or a function.

**Example:**

```cpp
double Average(const int a[], int size)
{
    int sum = 0;
    for (int i = 0; i < size; ++i)
        sum += a[i];
    return static_cast<double>(sum) / size;
}
```

```cpp
double Average(const double a[], int size)
{
    double sum = 0.0;
    for (int i = 0; i < size; ++i)
        sum += a[i];
    return (sum / size);
}
```

Overloading Functions

The following code shows how `Average()` is invoked:

```cpp
int main()
{
    int a[5] = {1, 2, 3, 4, 5};
    double b[5] = {1.1, 2.2, 3.3, 4.4, 5.5};
    cout << Average(a, 5) << " int average" << endl;
    cout << Average(b, 5) << " double average" << endl;
    return 0;
}
```

Inlining

- C++ provides the keyword `inline` to preface a function declaration.
- For functions declared as inline, the compiler will generate inline code instead of function calls.
- Inline functions lead to increased time efficiency and decreased memory efficiency.

**Question:** How about recursive functions?

**Answer:** There are no recursive inline functions because they would require infinite memory.

Inlining has a similar effect as macro expansion:

**Inlining example:**

```cpp```
inline double cube(double x)
{
    return (x * x * x);
}
```

**Macro example:**

```cpp```
#define CUBE(X)   ((X)*(X)*(X))
```

So what is the difference between inlining and macro expansion?

**Answer:** Macro expansion provides no type safety as is given by the C++ parameter-passing mechanism.

Therefore, it is advisable to use inlining instead of macro expansion.