COMP 2012H Honors Object-Oriented Programming and Data Structures

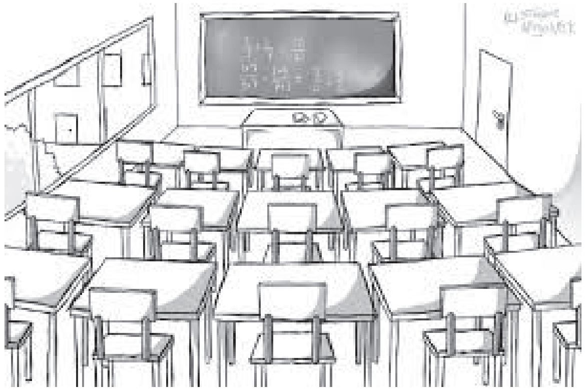
Topic 10: Class and Object

Dr. Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

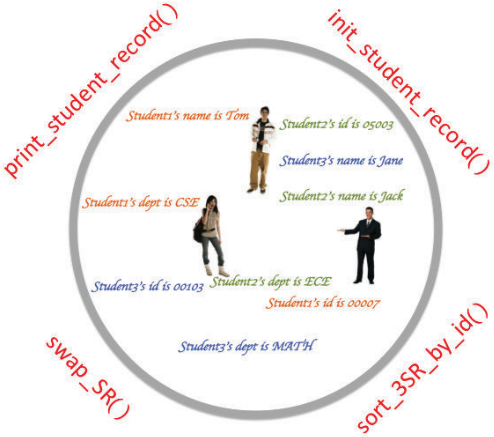
Part I

What is a C++ Class?



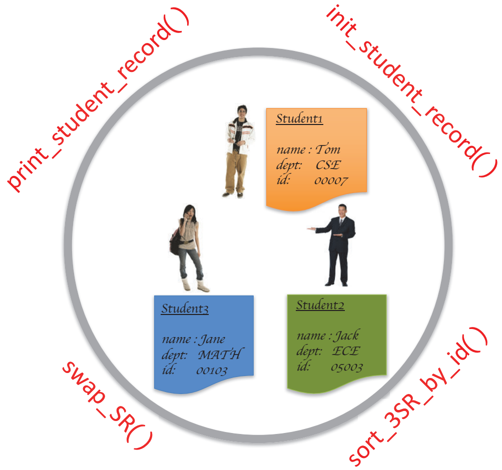
What Happens Before We Have C++ Class?

- Pieces of information, even belonging to the same object, are scattered around.
- All functions are global and are created to work on data.



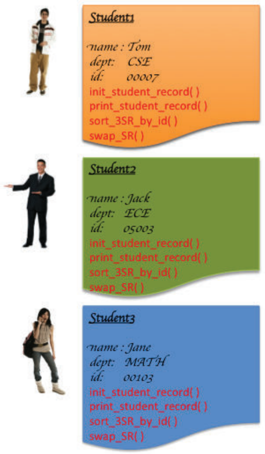
struct Helps Organize Data Better

- Pieces of information that belong to the same object are collected and wrapped in a struct.
- All functions are still global and are created to work on structs.



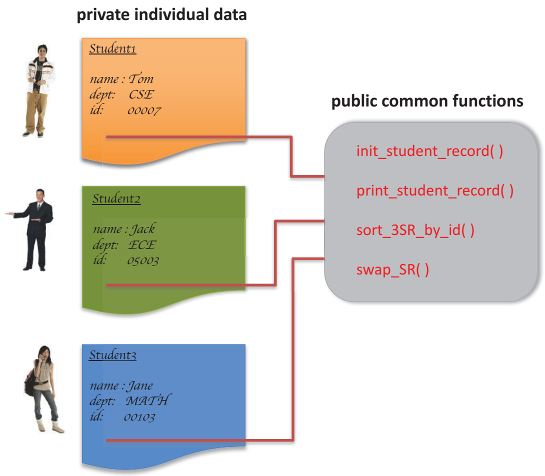
Perhaps We May Wrap Functions into struct as Well ???

- Functions are not global anymore. However, the function codes of different objects of the same struct are the same.
- Aren't the duplicate functions a waste?



Actual C++ Class Implementation

- Factor out the common functions so that the compiler generates only one copy of machine codes for each function.
- But functions are "struct-specific" — they can only be called by objects of the intended struct. Now you get a class!



C++ Class

- C++ struct allows you to create new complex data type consisting of a collection of generally heterogeneous objects.
- However, a basic data type like int, besides having a value, also supports a set of operations: $+$, -, $\times$, $/$, $\%$, $\gg$ (for input), and $\ll$ (for output).
- struct is inherited from the language C, and C++ generalizes the idea to class:

C++ Class

Class = data + operations + access control

or, Class = data member + member functions + access control

- Class allows you to create "smart" objects that support a set of operations.
- Class is also known as abstract data type (ADT).

C++ Class ..

- data members: just like data members in a struct.
- member functions: a set of functions that work only for the objects of the class, and can only be called by them.
- In addition, C++ allows access control to each data member and member function:
  - public: accessible to any functions (both member functions of the class or other functions)
  - private: accessible only to member functions of the class $\Rightarrow$ enforce information hiding
  - protected (will be discussed when we talk about inheritance)

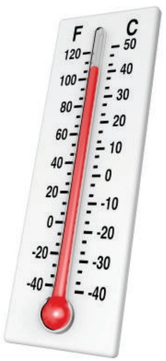
Example: Simplified student_record Class Definition

```
const int MAX_NAME_LEN = 32;

class student_record    /* File: student-record.h */
{
  private:
    char gender;
    char name[MAX_NAME_LEN];
    unsigned int id;

  public:
    // ACCESSOR member functions: const => won't modify data members
    const char* get_name() const { return name; }
    unsigned int get_id() const { return id; }
    void print() const
         { cout << name << endl << id << endl << gender << endl; }

    // MUTATOR member functions
    void set(const char my_name[], unsigned int my_id, char my_gender)
         { strcpy(name, my_name); id = my_id; gender = my_gender; }
    void copy(const student_record& r) { set(r.name, r.id, r.gender); }
};
```


Example: student-record-test.cpp

```
#include <iostream>    /* File: student-record-test.cpp */
using namespace std;
#include "student-record.h"

int main()
{
    student_record amy, bob; // Create 2 static student_record objects

    amy.set("Amy", 12345, 'F');    // Put values to their data members
    bob.set("Bob", 34567, 'M');

    cout << amy.get_id() << endl; // Get and print some data member
    amy.copy(bob);           // Amy want to fake Bob's identity
    amy.print();

    return 0;
    // Amy and Bob are static object, which will be destroyed
    // at the end of the function --- main() here --- call.
}   /* To compile: g++ student-record-test.cpp */
```


Example: A C++ Class for Temperature



Example: temperature Class Definition

```
#include <iostream>     /* File: temperature.h */
#include <cstdlib>
using namespace std;
const char CELSIUS = 'C', FAHRENHEIT = 'F';

class temperature
{
  private:
    char scale;
    double degree;
  public:
    // CONSTRUCTOR member functions
    temperature();     // Default constructor
    temperature(double d, char s);

    // ACCESSOR member functions: don't modify data
    char get_scale() const;
    double get_degree() const;
    void print() const;

    // MUTATOR member functions: will modify data
    void set(double d, char s);
    void fahrenheit(); // Convert to the Fahrenheit scale
    void celsius();    // Convert to the Celsius scale
};
```

Example: temperature Class Constructors

```cpp
/* File: temperature_constructors.cpp */
#include "temperature.h"


/* CONSTRUCTOR member functions */

// Default constructor
temperature::temperature()
{
    degree = 0.0;
    scale = CELSIUS;
}

// A general constructor
temperature::temperature(double d, char s)
{
    set(d, s);          // Calling another member function
}
```


Example: temperature Class Accessors

```cpp
/* File: temperature_accessors.cpp */
#include "temperature.h"

// ACCESSOR member functions
char temperature::get_scale() const
{
    return scale;
}

double temperature::get_degree() const
{
    return degree;
}

void temperature::print() const
{
    cout << degree << " " << scale;
}
```


Example: temperature Class Mutators

```cpp
#include "temperature.h" /* File: temperature_mutators.cpp */

void temperature::set(double d, char s)
{
    degree = d; scale = toupper(s); // lower case -> upper case

    if (scale != CELSIUS  &&  scale != FAHRENHEIT)
    { cout << "Bad temperature scale: " << scale << endl; exit(-1); }
}

void temperature::fahrenheit() // Conversion to the Fahrenheit scale
{
    if (scale == CELSIUS)
    { degree = degree*9.0/5.0 + 32.0; scale = FAHRENHEIT; }
}

void temperature::celsius()    // Conversion to the Celsius scale
{
    if (scale == FAHRENHEIT)
    { degree = (degree - 32.0)*5.0/9.0; scale = CELSIUS; }
}
```


Example: Testing the temperature Class

```cpp
#include "temperature.h" /* File: temperature_test.cpp */

int main()
{
    char scale;
    double degree;

    temperature x;            // Use default constructor
    x.print(); cout << endl; // Check the default values

    cout << "Enter temperature (e.g., 98.6 F): ";
    while (cin >> degree >> scale)
    {
        x.set(degree, scale);
        x.fahrenheit(); x.print(); cout << endl;// Convert to Fahrenheit
        x.celsius(); x.print(); cout << endl;   // Convert to Celsius
        cout << endl << "Enter temperature (e.g., 98.6 F): ";
    };

    return 0;
}
```

Constructors and Destructors

- An object is constructed when it is
  - defined in a scope (static object on stack).
  - created with the new operator (dynamic object on heap).
- An object is destructed when
  - it goes out of a scope (static object).
  - it is deleted with the delete operator (dynamic object).
- For "objects" (actually they are not objects in C++) of basic data types, their construction and destruction are built into the C++ language.
- For (real) objects of user-defined classes, C++ allows the class developers to write their own construction and destruction functions: constructors and destructor.
- Besides creating an object, a constructor may also initialize its contents.
- A class may have more than 1 constructor (function overloading), but it can only have 1 and only 1 destructor.

Default Constructors and Destructors

- If you do not provide a constructor/destructor, C++ will automatically generate the default constructor/destructor for you.
- The default constructor just reserves an amount of memory big enough for an object of the class.
  - no initialization will take place.
- The default destructor just releases the memory acquired by the object.

```
temperature::temperature() { }
temperature::~temperature() { }
```

- However, for a class that contains dynamic data members, the default constructors and destructors are usually inadequate, as they will not create and delete the dynamic data members for you.

Contents and Interface

- The data members are the contents of the objects of a class.
  - Usually they are made private.
  - Different objects of a class usually have different contents — different values for their data members.
- The member functions represent the interface to the objects of a class.
  - Usually they are made public.
  - private member functions are for internal use only.
  - Different objects of a class have the same interface!
- Both data members and member functions are members of a class. They are uniformly accessed by the . operator.
- An application programmer should
  - only use the public interface provided by the class developer to manipulate objects of a class.
  - a good class design will prevent the application programmer from accessing and modifying the data members directly.

Class Member Functions

- There are at least 4 types of member functions:
  - constructor : used to create class object.
  - destructor : used to destruct class objects. It is needed only when objects contain dynamic data member(s).
  - accessor : const functions that inspect data members; they do not modify any data members though.
  - mutator : will modify some data member(s).
- The member functions may be defined
  - inside the class definition in a .h header file.
  - outside the class definition in a .cpp source file. In that case, each function name must be prepended with the class name and the special class scope operator ::.

    ⇒ This is preferred so that application programmers won't see the implementation of the functions, and they are only given the .o object file of the class for development.

    ⇒ information hiding; protecting intellectual property.

To Enforce Information Hiding In Practice

- Developer of the class named "myclass" will only provide
  1. myclass.h: class definition header file
  2. libmyclass.a: library file that contains the object codes of the implementation of all class member functions (usually written in several source files)
- Using the temperature class as an example, one may build its library named "libtemperature.a" on Linux as follows:

```
g++ -c temperature_constructors.cpp
g++ -c temperature_accessors.cpp
g++ -c temperature_mutators.cpp
ar rsuv libtemperature.a temperature_constructors.o \
         temperature_accessors.o temperature_mutators.o
```

- An application programmer compiles an app named "temperature_test" with the source file "temperature_test.cpp" as follows:

```
g++ -o temperature_test temperature_test.cpp -L. -ltemperature
(or, g++ -o temperature_test temperature_test.cpp libtemperature.a)
```


Information Hiding Rules

1. DON'T expose data items in a class.
   ⇒ make all data members private.
   ⇒ class developer should maintain integrity of data members.
2. DON'T expose the difference between stored data and derived data.
   ⇒ the value that an accessor member function returns may NOT be the value of any data member. It is NOT necessary to have an accessor member function for each data member.
3. DON'T expose a class' internal structure.
   ⇒ application programmers should NOT assume the data structure used for data members.
   ⇒ class developer may change representation of data members without affecting the application programmers' codes.
4. DON'T expose the implementation details of a class.
   ⇒ class developer may change algorithm of member functions without affecting the application programmers' codes.

Example: Better Design of class temperature

- The last design of the class temperature changes its internal data, {degree, scale}, when the user wants the temperature in different scales.
- Thus, the information hiding rules 2 and 3 are not strongly followed.
- In a better design, the user does not need to know what the inside representation of {degree, scale} is when he wants to get the temperature in various scales.
- Let's re-design the temperature class to follow the last 4 rules of information hiding more closely.

Example: temperature Class Definition (2)

```
#include <iostream>     /* File: temperature.h */
#include <cstdlib>
using namespace std;
const char KELVIN = 'K', CELSIUS = 'C', FAHRENHEIT = 'F';

class temperature
{
  private:
    double degree;      // Internally it is always saved in Kelvin
  public:
    // CONSTRUCTOR member functions
    temperature();      // Default constructor
    temperature(double d, char s);

    // ACCESSOR member functions: don't modify data
    double kelvin() const;      // Read temperature in Kelvin
    double celsius() const;     // Read temperature in Fahrenheit
    double fahrenheit() const;  // Read temperature in Celsius

    // MUTATOR member functions: will modify data
    void set(double d, char s);
};
```

Example: temperature Class Constructors & Accessors (2)

```
/* File: temperature_constructors_accessors.cpp */
#include "temperature.h"

// CONSTRUCTOR member functions
temperature::temperature() { degree = 0.0; }
temperature::temperature(double d, char s) { set(d, s); }

// ACCESSOR member functions
double temperature::kelvin() const { return degree; }
double temperature::celsius() const { return degree - 273.15; }
double temperature::fahrenheit() const
                    { return (degree - 273.15)*9.0/5.0 + 32.0; }
```


Example: temperature Class Mutators (2)

```
#include "temperature.h" /* File: temperature_mutators.cpp */

void temperature::set(double d, char s)
{
    switch (s)
    {
        case KELVIN: degree = d; break;
        case CELSIUS: degree = d + 273.15; break;
        case FAHRENHEIT: degree = (d - 32.0)*5.0/9.0 + 273.15; break;

        default: cerr << "Bad temperature scale: " << s << endl;
            exit(-1);
    }

    if (degree < 0.0)   // Check for integrity of data
    {
        cerr << "Temperature less than absolute zero!" << endl;
        exit(-2);
    }
}
```


Example: Testing the temperature Class (2)

```
#include "temperature.h" /* File: temperature_test.cpp */

int main()
{
    char scale;
    double degree;
    temperature x;      // Use the default constructor

    cout << "Enter temperature (e.g., 98.6 F): ";
    while (cin >> degree >> scale)
    {
        x.set(degree, scale);
        cout << x.kelvin() << " K" << endl;     // Print in Kelvin
        cout << x.celsius() << " C" << endl;    // Print in Celsius
        cout << x.fahrenheit() << " F" << endl; // Print in Fahrenheit

        cout << endl << "Enter temperature (e.g., 98.6 F): ";
    };

    return 0;
}
```


OOP to Maintain Data/State Consistency/Integrity

- Data/State Consistency: Each time we change the value of degree in a temperature object, make sure that the new value is valid, since not all temperature values are possible.
- For a bigger object with many data members, changing the value of one member may affect other members.
- A snapshot of the values of all data members of an object represents the state of the object.
- Applications could easily set invalid or nonsensical values, causing a system to crash. OOP forces an application to use the API that performs sanity checks on all values to maintain consistent states of objects.
- Ensuring data/state consistency is one of the major challenges in (large) software projects.
- The problem becomes even more difficult when the prohgram is modified and new constraints are added.

Problem with non-OOP Implementation of Temperature

```
#include <iostream> /* File: non-oop-temperature.cpp */
using namespace std;
const char CELSIUS = 'C', FAHRENHEIT = 'F';

struct temperature
{
    char scale;
    double degree;
};

int main()
{
    temperature x;
    x.scale = CELSIUS;
    x.degree = -1000;      // That is IMPOSSIBLE!!!!
                          // But how can you prevent this to happen?
    return 0;
};
```

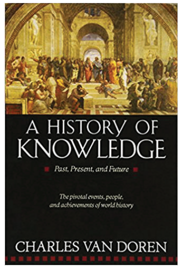

Summary: Classes and Objects

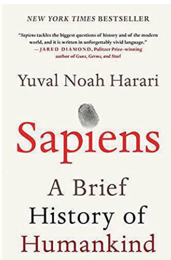
- A class is a user-defined type representing a set of objects with the same structure and behavior.
- Objects are variables of a class type.
- Instantiation: The process of creating an object of a class is called instantiating an object.
- Each object of a class has its own copies and values of its data members.
- All objects of a class share a common set of member functions.
- To call a function, before we say

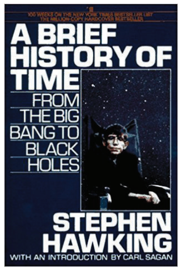
  "call function X" or simply "call X"

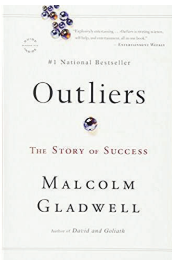
- In OOP, we have to say
  "invoke method/operation/function X on object Y of class Z"

Example: A C++ Class for Books









Example: Class with Dynamic Data Members — book.h

```
#include <iostream>     /* File: book.h */
using namespace std;
class Book              // Class definition written by class developer
{
  private:
    char* title;
    char* author;
    int num_pages;

  public:
    Book(int n = 100) { title = author = nullptr; num_pages = n; }
    Book(const char* t, const char* a, int n = 5) { set(t, a, n); }
    ~Book()
    {
        cout << "Delete the book titled \"" << title << "\"" << endl;
        delete [] title; delete [] author;
    }

    void set(const char* t, const char* a, int n)
    {
        title = new char [strlen(t)+1]; strcpy(title, t);
        author = new char [strlen(a)+1]; strcpy(author, a);
        num_pages = n;
    }
};
```

Example: Class with Dynamic Data Members — book.cpp

```
#include "book.h"       /* File: book.cpp */
void make_books()
{
    Book y("Love", "HKUST", 88);
    Book* p = new Book [3];
    p[0].set("book1", "author1", 1);
    p[1].set("book2", "author2", 2);
    p[2].set("book3", "author3", 3);

    delete [] p; cout << endl;
    return;
}

int main()    /* An app written by an application programmer */
{
    Book x("Sapiens", "Y. N. Harari", 1000);
    Book* z = new Book("Outliers", "Gladwell", 300);
    make_books(); cout << endl;
    delete z; cout << endl;
    return 0;
}
```


Example: Class with Dynamic Data Members — Output

```
Delete the book titled "book3"
Delete the book titled "book2"
Delete the book titled "book1"

Delete the book titled "Love"

Delete the book titled "Outliers"

Delete the book titled "Sapiens"
```

- Delete an array of user-defined objects using delete [].
- delete [] will call the class destructor on each array element in reverse order: the last element first till the first one.
- Notice also how the destructor of a static book is automatically called when it goes out of scope (e.g., when a function returns).

Part II

C++ Class Basics, Inline Functions & this Pointer

Structure vs. Class

In C++, structures are special classes and they may have member functions. By default,

$$\text{struct } \{ \ldots \} \equiv \text{class } \{ \text{ public: } \ldots \}$$
$$\text{class } \{ \ldots \} \equiv \text{struct } \{ \text{ private: } \ldots \}$$

```
#include <iostream> /* File: temperature-struct.h */
using namespace std;
const char KELVIN = 'K', CELSIUS = 'C', FAHRENHEIT = 'F';

struct temperature
{
    double degree;    // Internally it is always saved in Kelvin
    temperature();    // Default constructor
    temperature(double d, char s);
    double kelvin() const;     // Read temperature in Kelvin
    double celsius() const;    // Read temperature in Fahrenheit
    double fahrenheit() const; // Read temperature in Celsius
    void set(double d, char s);
};
```

Class Name: Name Equivalence

- A class definition introduces a new abstract data type.
- C++ relies on name equivalence (and not structure equivalence) for class types.

```
class X { int a; };
class Y { int a; };
class W { int a; };
class W { int b; };  // Error, double definition

X x;
Y y;

x = y;  // Error: type mismatch
```


Class Data Members

Data members can be any basic type, or any user-defined types if they are already declared.

Below are special cases:

- A class name can be used inside its own definition for a pointer to an object of the class:

```
class Cell
{
    int info;
    Cell* next;
};
```


Class Data Members ..

- A forward declaration of a class X can be used in the definition of another class Y to define a pointer to X:

```
class Cell;        // Forward declaration of Cell

class List
{
    int size;
    Cell* data;    // Points to a (forward-declared) Cell object
    Cell x;        // Error: Cell not defined yet!
};

class Cell         // Definition of Cell
{
    int info;
    Cell* next;
};
```


Default Initializer for Non-static Members (C++11)

```
class Complex
{
  private:
    float real = 1.3;    // Note: not allowed before C++11
    float imag {0.5};    // Use either = or { } initializer
  public:
    ...
};
```

- You are advised to initialize non-static data member values by
  - class constructors
  - class member initialization list in a constructor
  - class member functions
- Non-static data members that are not initialized by the 3 ways above will have the values of their default member initializers if they exist, otherwise their values are undefined.
- We'll talk about static vs. non-static members later. All data members you'll see most of the time are non-static.

Class Member Functions

- These are the functions declared inside the body of a class.
- They can be defined in two ways:

1. Within the class body, then they are inline functions. The keyword inline is optional in this case.

```
class temperature
{
    ...
    double kelvin() const { return degree; }
    double celsius() const { return degree - 273.15; }
};
```

Or,

```
class temperature
{
    ...
    inline double kelvin() const { return degree; }
    inline double celsius() const { return degree - 273.15; }
};
```


Class Member Functions ..

2. Outside the class body, then add the prefix consisting of the class name and the class scope operator ::
(Any benefits of doing this?)

```
/* File: temperature.h */
class temperature
{   ...
    double kelvin() const ;
    double celsius() const;
};

/* File: temperature.cpp */
#include "temperature.h"
...
double temperature::kelvin() const { return degree; }
double temperature::celsius() const { return degree - 273.15; }
```

Question: Can we add data and function declarations to a class after the end of the class definition?

Class Scope and Scope Operator ::

- C++ uses lexical (static) scope rules: the binding of name occurrences to declarations are done statically at compile-time.
- Identifiers declared inside a class definition are under its scope.
- To define the members functions outside the class definition, prefix the identifier with the class scope operator ::
- e.g., `temperature::kelvin()`, `temperature::celsius()`

```
int height = 10;
class Weird
{
    short height;
    Weird() { height = 5; }
};
```

Q1 : Which "height" is used in Weird::Weird( )?

Q2 : Can we access the global height inside the Weird class body?

Inline Functions

- Function calls are expensive because when a function is called, the operating system has to do a lot of things behind the scene to make that happens.

```
int f(int x) { return 4*x*x + 9*x + 1; }
int main() { int y = f(5); }
```

- For small functions that are called frequently, it is actually more efficient to unfold the function codes at the expense of program size (both source file and executable).

```
int main() { int y = 4*5*5 + 9*5 + 1; }
```

Inline Functions ..

- But functions have the benefit of easy reading, easy maintenance, and type checking by the compiler.
- You have the benefits of both by declaring the function inline.

```
inline int f(int x) { return 4*x*x + 9*x + 1; }
int main() { int y = f(5); }
```

- *However*, C++ compilers may not honor your inline declaration.
- The inline declaration is just a hint to the compiler which still has the freedom to choose whether to inline your function or not, especially when it is large!

Inline Class Member Functions

- Class member functions defined inside the class definition body are automatically treated as inline functions.
- To enhance readability, one may also define them outside the class definition but in the same header file.

```
/* File: temperature1.h */
class temperature
{   ...
    inline double
    kelvin() const
    {
        return degree;
    }
};
```

```
/* File: temperature2.h */
class temperature
{   ...
    inline double kelvin() const;
};

inline double
temperature::kelvin() const
{
    return degree;
}
```


Member Access Control

A member of a class can be:

1. public: accessible to anybody (class developer and application programmers)
2. private: accessible only to
   - member functions, and
   - friends of the class (decided by the class developer)

   ⇒ class developer enforces information hiding
3. protected: accessible to
   - member functions and friends of the class, as well as
   - member functions and friends of its derived classes (subclasses)

   ⇒ class developer restricts what subclasses may directly use (more about this when we talk about inheritance)

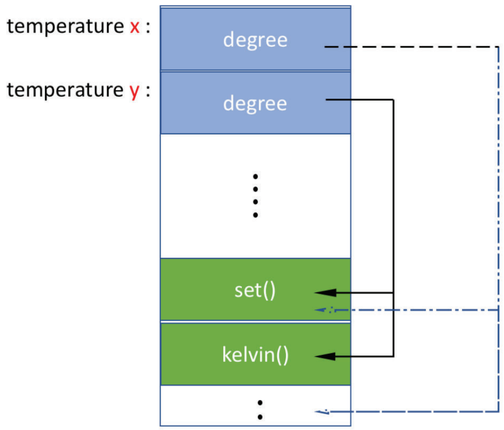
Example: Member Access Control

```
class Stack              /* File: access_control.cpp */
{
  private:
    int data[BUFFER_SIZE];
    int top_index;
  public:
    void push(int);
    ...
};

int main()
{
    Stack x;
    x.push(2);           // OK: push( ) is public
    cout << x.top_index; // Error: cannot access top_index
    return 0;
}
```

How Are Objects Implemented?

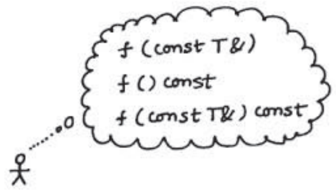
- Each class object gets its own copy of the class data members.
- All objects of the same class share one copy of the member functions.

```
int main()
{
    temperature x, y;

    x.set(10, CELSIUS);
    y.set(30, KELVIN);
    return 0;
}
```



Part III

this Pointer & const-ness



this Pointer

- Each class member function implicitly contains a pointer of its class type named "this".
- When an object calls the function, this pointer is set to point to the object.
- For example, after compilation, the `temperature::set(double d, char s)` function in the temperature class will be translated to a unique global function by adding a new argument:

```
void temperature::set(temperature* this, double d, char s)
{
    switch (s)
    {
        case KELVIN: this->degree = d; break;
        case CELSIUS: this->degree = d + 273.15; break;
        ...
    }
    ...
}
```

- x.set(d, s) becomes temperature::set(&x, d, s).

const this pointer

- For const member functions, their implicit this pointer is a const pointer.
- That is, the this pointer in a const member functions is pointing to a const object.
- Thus, the object can't be modified inside a const member function.
- E.g., after compilation,

  `temperature::kelvin() const`

  will be translated to a global function like:

```
void temperature::kelvin(const temperature* this) const
{
    return this->degree;
}
```

Example: Return an Object by this — RBR

```
class Complex           /* File: complex.h */
{
  private:
    float real; float imag;

  public:
    Complex(float r, float i) { real = r; imag = i; }
    void print() { cout << "(" << real << " , " << imag << ")" << endl; }

    Complex add1(const Complex& x)  // Return by value
    {
        real += x.real; imag += x.imag;
        return (*this);
    }
    Complex* add2(const Complex& x) // Return by value using pointer
    {
        real += x.real; imag += x.imag;
        return this;
    }
    Complex& add3(const Complex& x) // Return by reference
    {
        real += x.real; imag += x.imag;
        return (*this);
    }
};
```


Example: Return an Object by this — RBR ..

```
#include <iostream>     /* File: complex-test.cpp */
using namespace std;
#include "complex.h"

int main()
{
    Complex y(3, 4); y.print();

    cout << endl << "Return by value" << endl;
    Complex x(1, 2); x.print();
    x.add1(y).add1(y).print();
    x.print();

    cout << endl << "Return its pointer by value" << endl;
    x.add2(y)->print();
    x.print();

    cout << endl << "Return by reference" << endl;
    Complex z(1, 2); z.print();
    z.add3(y).add3(y).print();
    z.print();

    return 0;
}
```


Return by Value and Return by Reference

There are 2 ways to pass parameters to a function

- pass-by-value (PBV)
- pass-by-reference (PBR)
  - lvalue reference: that is what you learned in the past and we'll keep just saying *reference* for lvalue reference.
  - rvalue reference (C++11)

Similarly, you may return from a function by returning an object's

- value: the function will make a separate copy of the object and return it. Changes made to the copy have no effect on the original object.
- (lvalue) reference: the object itself is passed back! Any further operations on the returned object will directly modify the original object as it is the same as the returned object.
- rvalue reference: we'll talk about this later.

const

- const, in its simplest usage, is used to express a user-defined constant — a value that can't be changed.

  `const float PI = 3.1416;`
- Some people like to write const identifiers in capital letters.
- In the old days, constants are defined by the #define preprocessor directive:

  `#define PI 3.1416`

  Question: Any shortcomings?
- const actually may be used to represent more than just numerical constants, but also const objects, pointers, and even member functions!
- The const keyword can be regarded as a safety net for programmers: If an object should not change, make it const.

Example: Constants of Basic Types

```
#include <iostream>     /* File: const-basic-types.cpp */
#include <cstdlib>      // Needed for calling rand() etc.
using namespace std;

int main()  // To generate 5 random integers in [0, 100] inclusive
{
    const int N = 5;          // Number of integers to generate
    const int MAX_NUM = 100; // Max integer

    // Seed the random number generator by current time
    srand(time(0));

    for (int j = 0; j < N; ++j)
        cout << rand() % (MAX_NUM+1) << endl;

    return 0;
}
```


Example: Constant Object of User-defined Types

```
/* File: const-object-date.h */

class Date      // There are problems with this code; what are they?
{
  private:
    int year, month, day;

  public:
    Date() { cin >> year >> month >> day; }
    Date(int y, int m, int d) { year = y; month = m; day = d; }

    void add_month() { month += 1; }; // Will be an inline function

    int difference(const Date& d)
    { /* Incomplete: write this function */ }

    void print()
    { cout << year << "/" << month << "/" << day << endl; }
};
```


Example: Constant Object of User-defined Types ..

```
#include <iostream>     /* File: const-object-date.cpp */
using namespace std;
#include "const-object-date.h"

int main()   // There are problems with this code; what are they?
{
    const Date WW2(1945, 9, 2); // World War II ending date
    Date today;
    WW2.print();
    today.print();

    // How long has it been since World War II?
    cout << "Today is " << today.difference(WW2)
         << " days after WW2" << endl;

    // What about next month?
    WW2.add_month();    // Error; do you mean today.add_month()??
    cout << today.difference(WW2) << " days by next month.\n";

    return 0;
}
```


const Member Functions

- To indicate that a class member function does not modify the class object — its data member(s), one can (and should!) place the const keyword after the argument list.

```
class Date              /* File: const-object-date2.h */
{
  private:
    int year, month, day;

  public:
    Date() { cin >> year >> month >> day; }
    Date(int y, int m, int d) { year = y; month = m; day = d; }
    void add_month() { month += 1; }; // Will be an inline function

    int difference(const Date& d) const { /* Incomplete */ }
    void print() const
    { cout << year << "/" << month << "/" << day << endl; }
};
```

const Member Functions and this Pointer

- A const object can only call const member functions of its class.
- But a non-const object can call both const and non-const member functions of its class.
- The this pointer in const member functions points to const objects. For example,

  ▷ `int Date::difference(const Date& d) const;` is compiled to

  ```
  int Date::difference(const Date* this, const Date& d);
  ```

  ▷ `void Date::print() const;` is compiled to

  ```
  void Date::print(const Date* this);
  ```

- Thus, the object calling const member function becomes const inside the function and cannot be modified.

const and const Pointers

- When a pointer is used, two objects are involved:
  - the pointer itself
  - the object being pointed to
- The syntax for pointers to constant objects and constant pointers can be confusing. The rule is that
  - any const to the left of the * in a declaration refers to the object being pointed to.
  - any const to the right of the * refers to the pointer itself.
- It can be helpful to read these declarations from right to left.

```
/* File: const-char-ptrs1.cpp */
char c = 'Y';
char *const cpc = &c;
char const* pcc;
const char* pcc2;
const char *const cpcc = &c;
char const *const cpcc2 = &c;
```


Example: const and const Pointers

```
#include <iostream>     /* File: const-char-ptrs2.cpp */
using namespace std;

int main()
{
    char s[] = "COMP2012H"; // Usual initialization in the past
    char p[] {"MATH1013"};  // C++11 style of uniform initialization

    const char* pcc {s};    // Pointer to constant char
    pcc[5] = '5';           // Error!
    pcc = p;                // OK, but what does that mean?

    char *const cpc = s;    // Constant pointer
    cpc[5] = '5';           // OK
    cpc = p;                // Error!

    const char *const cpcc = s; // const pointer to const char
    cpcc[5] = '5';          // Error!
    cpcc = p;               // Error!
    return 0;
}
```


const and const Pointers ..

- Having a point-to-const pointing to a non-const object doesn't make the object a constant!

```
/* File: const-int-ptr.cpp */
int i = 151;
i += 20;    // OK

int* pi = &i;
*pi += 20;  // OK

const int* pic = &i;
*pic += 20; // Error! Can't change i through pic

pic = pi;   // OK
*pic += 20; // Error! Can't change *pi thru pic

pi = pic;   // Error: Invalid conversion from 'const int*' to 'int*'
```

const References as Function Arguments

- There are 2 good reasons to pass an argument as a reference. What are they?
- You can (and should!) express your intention to leave a reference argument of your function unchanged by making it const.
- There are 2 advantages:

1. If you accidentally try to modify the argument in your function, the compiler will catch the error.

```
void cbr(int& x) { x += 10; }          // Fine

void cbcr(const int& x) { x += 10; } // Error!
```


const References as Function Arguments ..

2. You may pass both const and non-const arguments to a function that requires a const reference parameter.

   Conversely, you may pass only non-const arguments to a function that requires a non-const reference parameter.

```
#include <iostream>
using namespace std;
void cbr(int& a) { cout << a << endl; }
void cbcr(const int& a) { cout << a << endl; }
int main()
{
    int x {50}; const int y {100};
    // Which of the following give(s) compilation error?
    cbr(x);
    cbcr(x);
    cbr(y);
    cbcr(y);
    cbr(1234);
    cbcr(1234);
}
```


Summary: Good Practice

- Objects you don't intend to change ⇒ const objects

```
const double PI = 3.1415927;
const Date handover(1, 7, 1997);
```

- Function arguments you don't intend to change
  ⇒ const arguments

```
void print_height(const Large_Obj& LO){ cout << LO.height(): }
```

- Class member functions that don't change the data members
  ⇒ const member functions

```
int Date::get_day() const { return day; }
```


Summary

- Regarding which objects can call const or non-const member functions:

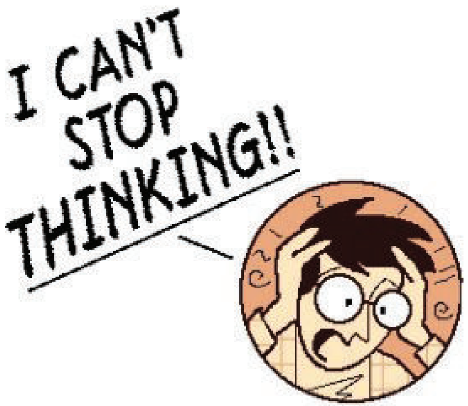
Calling Object	const Member Function	non-const Member Function
const Object	√	X
non-const Object	√	√

- Regarding which objects can be passed to functions with const or non-const arguments:

Passing Object	const Function Argument	non-const Function Argument
literal constant	√	X
const Object	√	X
non-const Object	√	√

That's all!

Any questions?



Further Reading

Example: 2 C++ Classes — Bulbs and Lamps

Example: Bulbs and Lamps

- This example consists of 2 classes: Bulb and Lamp.
- A lamp has at least one light bulb.
- All bulbs of a lamp are the same in terms of price and wattage (power).
- The price of a lamp that is passed to the Lamp's constructor does not include the price of its bulbs which have to be bought separately.
- One installs bulb(s) onto a lamp by calling its member function install_bulbs.

Example: lamp-test.cpp

```
#include "lamp.h" /* File: lamp-test.cpp */

int main()
{
    Lamp lamp1(4, 100.5); // lamp1 costs 100.5 and needs 4 bulbs
    Lamp lamp2(2, 200.6); // lamp2 costs 200.6 and needs 2 bulbs

    // Install 4 bulbs of 20 Watts, each costing 30.1 on lamp1
    lamp1.install_bulbs(20, 30.1);
    lamp1.print("lamp1");

    // Install 2 bulbs of 60 Watts, each costing 50.4 on lamp2
    lamp2.install_bulbs(60, 50.4);
    lamp2.print("lamp2");

    return 0;
}
/* To compile: g++ -o lamp-test lamp-test.cpp bulb.cpp lamp.cpp */
```


Example: bulb.h

```
/* File: bulb.h */

class Bulb
{
  private:
    int wattage;        // A light bulb's power in watt (W)
    float price;        // A light bulb's price in dollars

  public:
    int get_power() const;
    float get_price() const;
    void set(int w, float p); // w = bulb's wattage; p = its price
};
```


Example: bulb.cpp

```
/* File: bulb.cpp */

#include "bulb.h"

int Bulb::get_power() const { return wattage; }

float Bulb::get_price() const { return price; }

void Bulb::set(int w, float p) { wattage = w; price = p; }
```


Example: lamp.h

```
#include "bulb.h"       /* File: lamp.h */
class Lamp
{
  private:
    int  num_bulbs; // A lamp MUST have 1 or more light bulbs
    Bulb* bulbs; // Dynamic array of light bulbs installed onto a lamp
    float price; // Price of the lamp, not including its bulbs

  public:
    Lamp(int n, float p);      // n = number of bulbs; p = lamp's price
    ~Lamp();

    int total_power() const;   // Total power/wattage of its bulbs
    float total_price() const; // Price of a lamp PLUS its bulbs

    // Print out a lamp's information; see outputs from our example
    void print(const char* prefix_message) const;

    // All light bulbs of a lamp have the same power/wattage and price:
    // w = a light bulb's wattage; p = a light bulb's price
    void install_bulbs(int w, float p);
};
```

Example: lamp.cpp

```cpp
#include "lamp.h"       /* File: lamp.cpp */
#include <iostream>
using namespace std;

Lamp::Lamp(int n, float p)
          { num_bulbs = n; price = p; bulbs = new Bulb [n]; }
Lamp::~Lamp() { delete [] bulbs; }
int Lamp::total_power() const
          { return num_bulbs*bulbs[0].get_power(); }
float Lamp::total_price() const
          { return price + num_bulbs*bulbs[0].get_price(); }
void Lamp::print(const char* prefix_message) const
{
    cout << prefix_message << ": total power = " << total_power()
         << "W" << " , total price = $" << total_price() << endl;
}
void Lamp::install_bulbs(int w, float p)
{
    for (int j = 0; j < num_bulbs; ++j)
        bulbs[j].set(w, p);
}
```


Example: Linked-list Char Node Class Definition

```cpp
#include <iostream>     /* File: ll_cnode.h */
using namespace std;
const char NULL_CHAR = '\0';

class ll_cnode
{
  public:
    char data;      // Single character node
    ll_cnode* next; // The link to the next character node

   // CONSTRUCTOR
    ll_cnode(char c = NULL_CHAR) { data = c; next = nullptr; }
};
```


Example: mystring Class Constructors Using LL

```cpp
#include "mystring.h"   /* File: mystring_constructors.cpp */

mystring::mystring() { head = nullptr; } // Default constructor

mystring::mystring(char c) { head = new ll_cnode(c); }

mystring::mystring(const char s[])
{
    if (s[0] == NULL_CHAR) // Empty linked list due to empty C string
    {
        head = nullptr; return;
    }

    // First copy s[0] to the first node of mystring
    ll_cnode* p = head = new ll_cnode(s[0]);

    // Add a new ll_cnode for each char in the char array s[]
    for (int j = 1; s[j] != NULL_CHAR; j++, p = p->next)
        p->next = new ll_cnode(s[j]);
    p->next = nullptr;  // Set the last ll_cnode to point to NOTHING
}
```


Example: mystring Class Accessor Functions Using LL

```cpp
#include "mystring.h"   /* File: mystring_accessors.cpp */

int mystring::length() const
{
    int length = 0;
    for (const ll_cnode* p = head; p != nullptr; p = p->next)
        length++;

    return length;
}


void mystring::print() const
{
    for (const ll_cnode* p = head; p != nullptr; p = p->next)
        cout << p->data;

    cout << endl;
}
```

Example: mystring Class Mutator Functions — insert( )

```
#include "mystring.h"   /* File: mystring_insert.cpp */
// To insert character c to the linked list so that after insertion,
// c is the n-th character (counted from zero) in the list.
// If n > current length, append to the end of the list.

void mystring::insert(char c, unsigned n)
{   // STEP 1: Create the new ll_cnode to contain char c
    ll_cnode* new_cnode = new ll_cnode(c);
    if (n == 0 || head == nullptr) // Special case: insert at the beginning
    {
        new_cnode->next = head;
        head = new_cnode;
        return;
    }
    // STEP 2: Find the node after which the new node is to be added
    ll_cnode* p = head;
    for (int position = 0;
        position < n-1  &&  p->next != nullptr;
         p = p->next, ++position)
         ;

    // STEP 3,4: Insert the new node between the found node and the next node
    new_cnode->next = p->next; // STEP 3
    p->next = new_cnode;       // STEP 4
}
```


Example: mystring Class Mutator Functions — remove( )

```
#include "mystring.h"   /* File: mystring_remove.cpp */
// To remove the character c from the linked list.
// Do nothing if the character cannot be found.
void mystring::remove(char c)
{
    ll_cnode* previous = nullptr; // Point to previous ll_cnode
    ll_cnode* current = head;  // Point to current ll_cnode
    // STEP 1: Find the item to be removed
    while (current != nullptr && current->data != c)
    {
        previous = current;    // Advance both pointers
        current = current->next;
    }

    if (current != nullptr)       // Data is found
    {
        // STEP 2: Bypass the found item
        if (current == head)   // Special case: Remove the first item
            head = head->next;
        else
            previous->next = current->next;
        // STEP 3: Free up the memory of the removed item
        delete current;
    }
}
```


Example: mystring Class Destructors Using LL

```
#include "mystring.h"   /* File: mystring_destructor.cpp */

mystring::~mystring()
{
    if (head == nullptr) // No need to do destruction for empty mystring
        return;

    ll_cnode* current;  // Point to current ll_cnode
    ll_cnode* next;     // Point to next ll_cnode

    // Go through the linked list and delete one node at a time
    for (current = head; current != nullptr; current = next)
    {
        next = current->next;
        delete current; // Free up the memory of each ll_cnode
    }
}
```


Change Internal Representation of Class mystring to Array

- The last design of the class mystring uses a linked-list of characters to represent a character string.
- The class developer later decides to change the representation to a character array. As a consequence, he also has to change the implementation of all the member functions.
- Thanks to the OOP approach, the class developer can do that without changing the public interface of the class mystring. As a consequence,
  - class developer has to give the new class definition header file,
  - and the new library file to the application programmer,
  - and the application programmer does not need to change their programs, but only re-compiles his programs with the new library.

Example: mystring Class Definition Using Array

```
#include <iostream>     /* File: mystring.h */
#include <cstdlib>
using namespace std;
const int MAX_STR_LEN = 1024; const char NULL_CHAR = '\0';

class mystring
{
  private:
    char data[MAX_STR_LEN+1];

  public:
    // CONSTRUCTOR member functions
    mystring();              // Construct an emtry string
    mystring(char);          // Construct from a single char
    mystring(const char[]); // Construct from a C-string
    // DESTRUCTOR member function
    ~mystring();
    // ACCESSOR member functions: Again declared const
    int length() const;
    void print() const;
    // MUTATOR member functions
    void insert(char c, unsigned n); // Insert char c at position n
    void remove(char c);       // Delete the first occurrence of char c
};
```


Example: mystring Constructors, Destructor, Accessors

```
/* File: mystring_constructors_destructor_accessors.cpp */
#include "mystring.h"
// Constructors
mystring::mystring() { data[0] = NULL_CHAR; }
mystring::mystring(char c) { data[0] = c; data[1] = NULL_CHAR; }
mystring::mystring(const char s[])
{
    if (strlen(s) > MAX_STR_LEN)
    {
        cerr << "mystring::mystring --- Only a max. of "
             << MAX_STR_LEN << " characters are allowed!" << endl;
        exit(1);
    }
    strcpy(data, s);
}

// Destructor
mystring::~mystring() { }

// ACCESSOR member functions
int mystring::length() const { return strlen(data); }
void mystring::print() const { cout << data << endl; }
```


Example: mystring Class Mutator — insert( ) Using Array

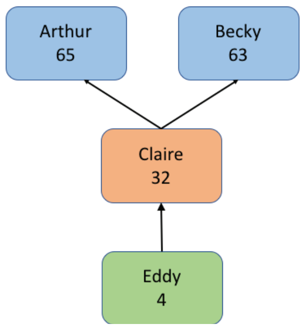
```
#include "mystring.h"   /* File: mystring_insert.cpp */

void mystring::insert(char c, unsigned n)
{
    int length = strlen(data);
    if (length == MAX_STR_LEN)
    {
        cerr << "mystring::insert --- string is already full!" << endl;
        exit(1);
    }

    int insert_position = (n >= length) ? length : n;

    for (int j = length; j != insert_position; j--)
        data[j] = data[j-1];

    data[insert_position] =  c;
    data[length+1] = NULL_CHAR;
}
```


Example: mystring Class Mutator — remove( ) Using Array

```
#include "mystring.h"   /* File: mystring_remove.cpp */

void mystring::remove(char c)
{
    int j;
    int mystring_length = length();

    for (j = 0; j < mystring_length; j++)
    {
        if (data[j] == c)
            break;
    }

    if (j < mystring_length)    // c is found
    {
        for (; j < mystring_length; j++)
            data[j] = data[j+1];
    }
}
```

Example: Person & Family

- It consists of a the class Person, from which families are built.
- A person, in general, has at most 1 child, and his/her father and mother may or may not be known.
- The information of his/her family includes him/her and his parents and grandparents from both of his/her parents.



Example: Expected Output

```
Name: Arthur
Father: unknown
Mother: unknown
Grand Fathers: unknown, unknown
Grand Mothers: unknown, unknown

Name: Becky
Father: unknown
Mother: unknown
Grand Fathers: unknown, unknown
Grand Mothers: unknown, unknown

Name: Claire
Father: Arthur
Mother: Becky
Grand Fathers: unknown, unknown
Grand Mothers: unknown, unknown

Name: Eddy
Father: unknown
Mother: Claire
Grand Fathers: unknown, Arthur
Grand Mothers: unknown, Becky
```

Example: Person Class — Header File

```
#include <iostream>     /* File: person.h */
using namespace std;

class Person
{
  private:
    char* _name;
    int _age;
    Person *_father, *_mother, *_child;

  public:
    Person(const char* my_name, int my_age, Person* my_father = nullptr,
           Person* my_mother = nullptr, Person* my_child = nullptr);
    ~Person();

    Person* father() const;
    Person* mother() const;
    Person* child() const;
    void have_child(Person* baby) ;
    void print_age() const;
    void print_name() const;
    void print_family() const;
};
```


Example: Person Class — Implementation File I

```
#include "person.h"     /* File: person.cpp */
#include <cstring>

Person::Person(const char* my_name, int my_age, Person* my_father,
               Person* my_mother, Person* my_child)
{
    _name = new char [strlen(my_name)+1];
    strcpy(_name, my_name);
    _age = my_age;
    _father = my_father;
    _mother = my_mother;
    _child = my_child;
};

Person::~Person() { delete [] _name; }

Person* Person::father() const { return _father; }

Person* Person::mother() const { return _mother; }

Person* Person::child() const { return _child; }

void Person::have_child(Person* baby) { _child = baby; }
```

Example: Person Class — Implementation File II

```
void Person::print_age() const { cout << _age; }

void Person::print_name() const
{
    cout << (_name ? _name : "unknown");
}

// Helper function
void print_parent(Person* parent)
{
    if (parent)
        parent->print_name();
    else
        cout << "unknown";
}
```


Example: Person Class — Implementation File III

```
void Person::print_family() const
{
    Person *f_grandfather = nullptr, *f_grandmother = nullptr,
        *m_grandfather = nullptr, *m_grandmother = nullptr;

    if (_father) {
        f_grandmother = _father->mother();
        f_grandfather = _father->father();
    }

    if (_mother) {
        m_grandmother = _mother->mother();
        m_grandfather = _mother->father();
    }

    cout << "Name: "; print_name(); cout << endl;
    cout << "Father: "; print_parent(_father); cout << endl;
    cout << "Mother: "; print_parent(_mother); cout << endl;

    cout << "Grand Fathers: "; print_parent(f_grandfather);
    cout << ", "; print_parent(m_grandfather); cout << endl;
    cout << "Grand Mothers: "; print_parent(f_grandmother);
    cout << ", "; print_parent(m_grandmother); cout << endl;
}
```


Example: Family Building Test Program

```
#include "person.h"     /* File: family.cpp */

int main()
{
    Person arthur("Arthur", 65, nullptr, nullptr, nullptr);
    Person becky("Becky", 63, nullptr, nullptr, nullptr);
    Person claire("Claire", 32, &arthur, &becky, nullptr);
    Person eddy("Eddy", 4, nullptr, &claire, nullptr);

    arthur.have_child(&claire);
    becky.have_child(&claire);
    claire.have_child(&eddy);

    arthur.print_family(); cout << endl;
    becky.print_family();  cout << endl;
    claire.print_family(); cout << endl;
    eddy.print_family();   cout << endl;
    return 0;
}
```