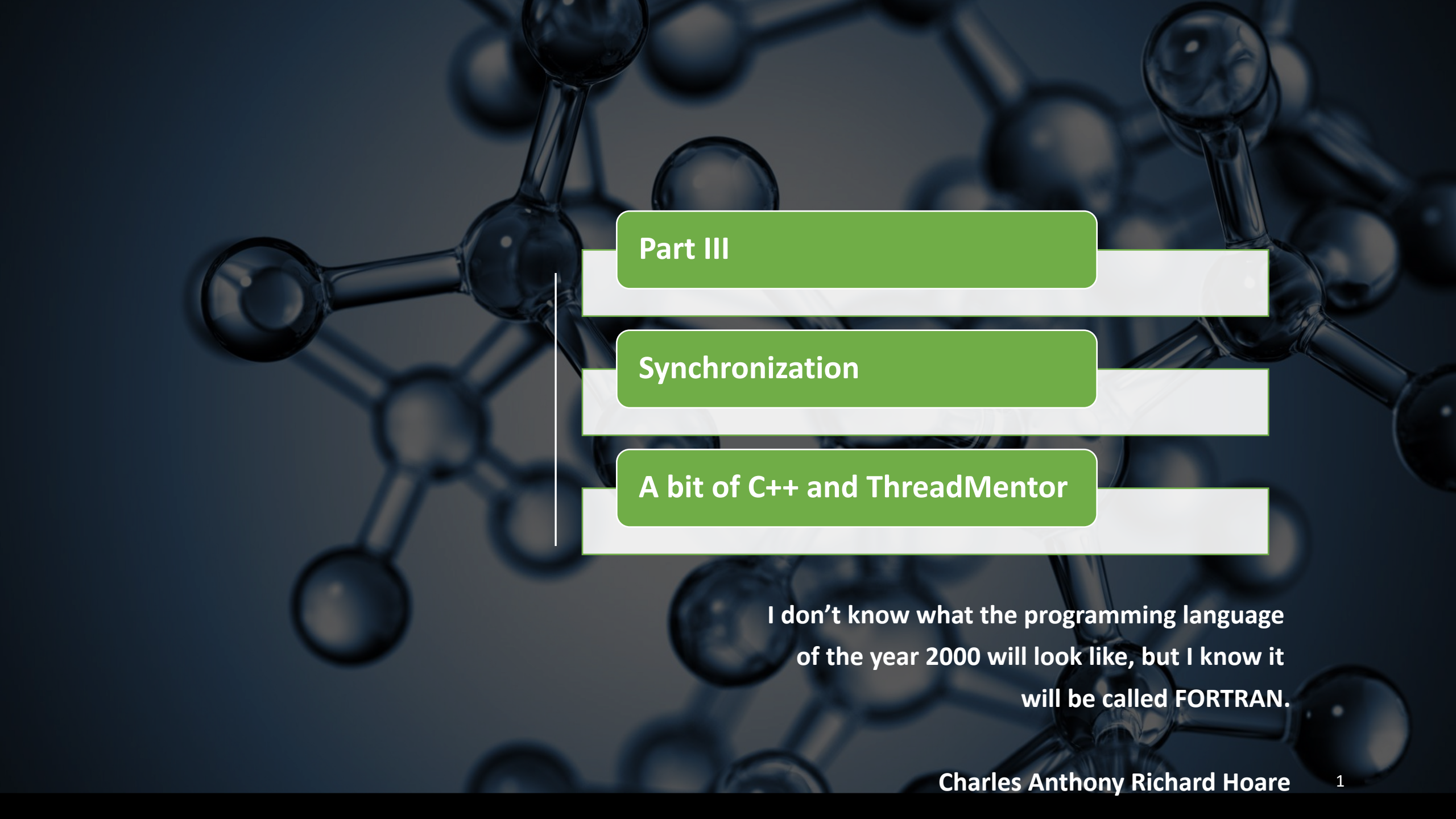


Part III

Synchronization

A bit of C++ and ThreadMentor

I don’t know what the programming language of the year 2000 will look like, but I know it will be called FORTRAN.

Charles Anthony Richard Hoare

Basics

- C++ is a superset of C
  - Anything that works in C will work in C++
  - C++ is object-oriented (like Java)
- Files are named **`name.cpp`** or **`name.cc`**
- GNU compiler is g++

- Only covering enough to use C++ thread package for this course

iostream and namespace

- Include `iostream` for input/output.
- Then, add `using namespace std;`

Namespaces allows control over the scope of otherwise global names; entities like classes, objects and functions to be grouped together into sub-scopes

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

int  main(…)
{
   // other C/C++ statements
}
```

Input with `cin` and >>

- Use `cin` and `>>` to read from `stdin`.
- For example, `cin >> n` reads in a data item from `stdin` to variable `n`.
- One more example: `cin >> a >> b` reads in two data items from `stdin` to variables `a` and `b` in this order.
- Thus, `cin` is easier to use than `scanf`.

Output with `cout` and `<<`

- Use `cout` and `<<` to write to `stdout`.
- For example, `cout << n` writes the content of variable `n` to `stdout`.
- One more example: `cout << a << b` writes the values of variables `a` and `b` to `stdout` in this order.
- Thus, `cout` is easier to use than `printf`.
- Formatted output with `cout` is tedious.

- The `\n` is `endl`: `cout << a << endl` prints the value of `a` and follows by a newline.
- You may want to add spaces to separate two printed values, in particular between two strings.
- The formatted output depends on some default setting of a system.
- `cout << a << ' ' << b << endl` is better than `cout << a << b << endl`.

cin/cout Example 1

hello.cpp

```
#include <iostream>

using namespace std;

int main(void)
{
    cout << "Hello, world." << endl;
    return 0;
}
```

cin/cout Example 2

factorial.cpp

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

int main(void){
   int  i, n, factorial;

   cout << "A positive integer --> ";
   cin  >> n;
   factorial = 1;
   for (i = 1; i <= n; i++) factorial *= i;
   cout << "Factorial of " << n << " = "
        << factorial << endl;
   return 0;
}
```

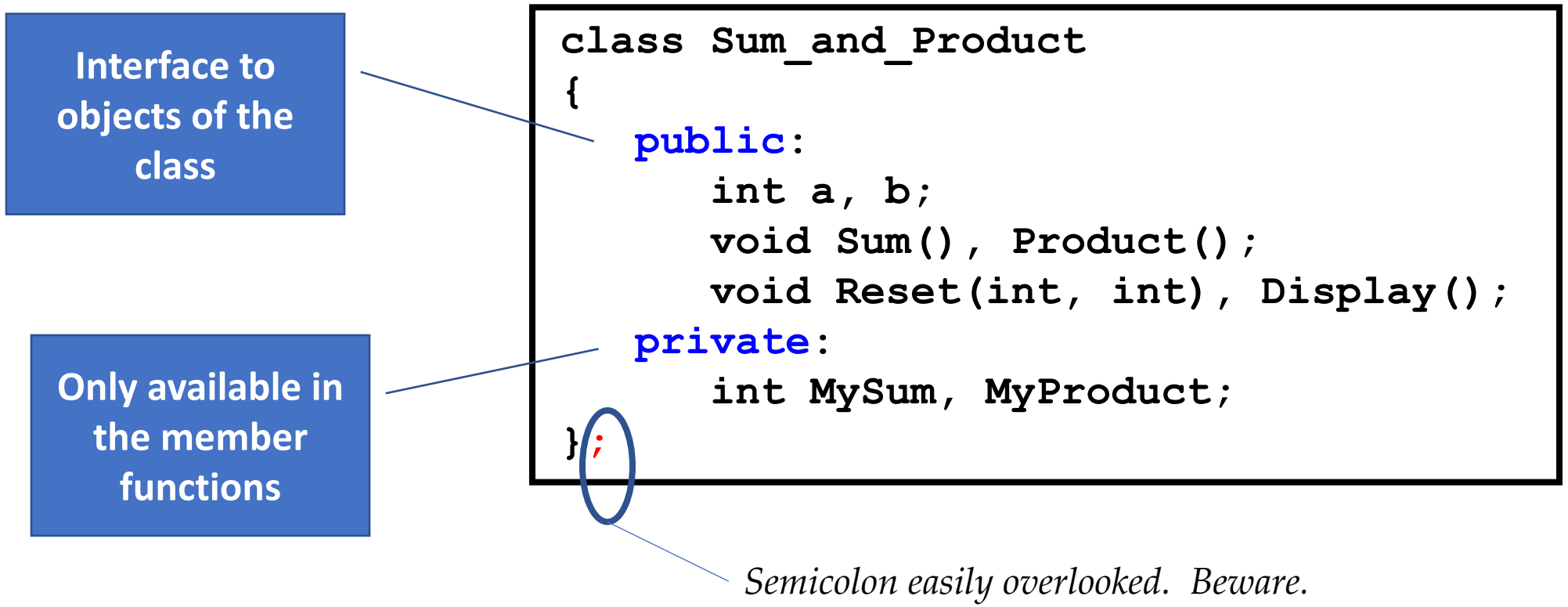
cin/cout Example 3

```
int main(void)
{
   int x = 1;
   int y = 12;
   float f = 3.4;

   cout << x << y << f << endl;

  return 0;
}
```

```
$./twoout
1123.4
```

What Is a `class`?

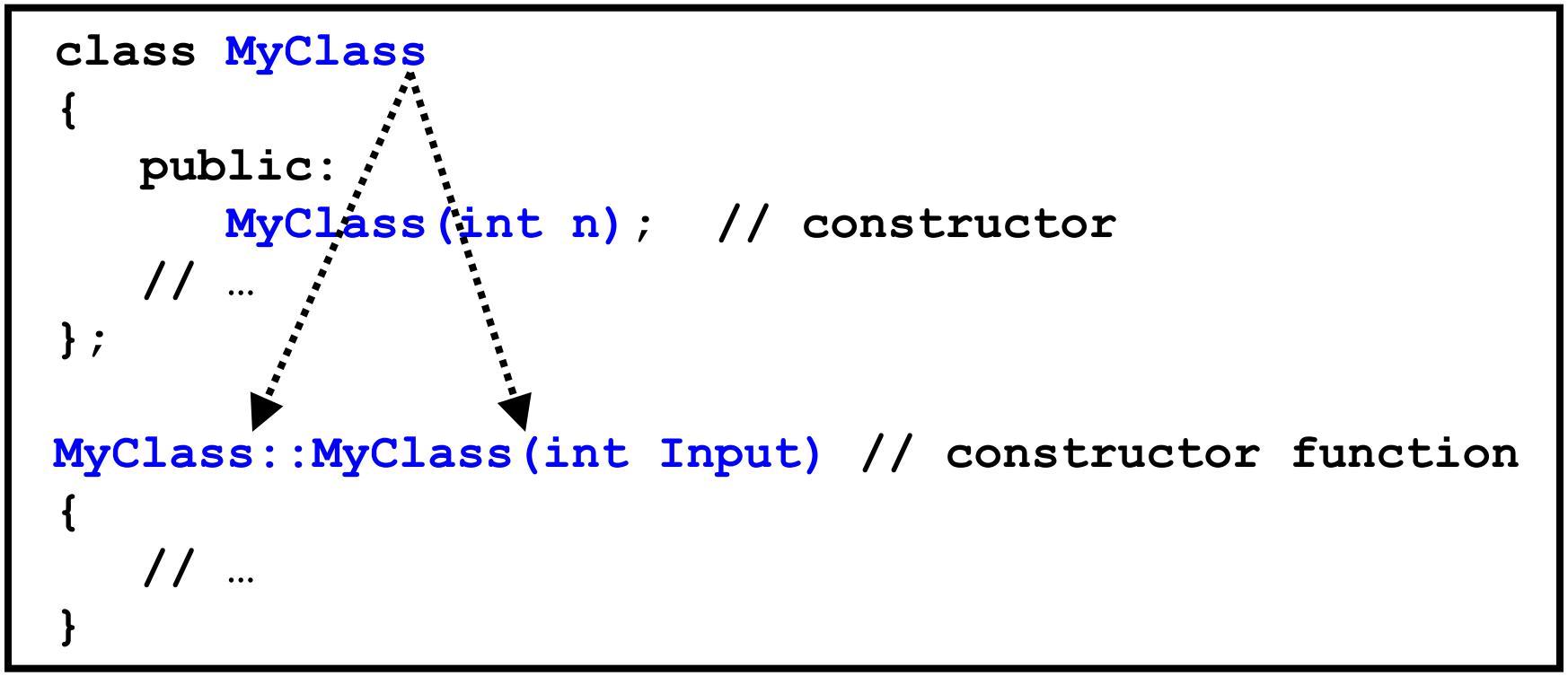
- A `class` is a type similar to a `struct`; but, a `class` type normally has member functions and member variables.



Constructors

- Constructors are member functions and are commonly used to initialize member variables in a class.
- A constructor is called when its class is created.
- A constructor has the same name as the class.
- A constructor definition **cannot** return a value, and no type, not even `void`, can be given at the beginning of the function or in the function header.

- **Constructors are commonly used to initialize member variables in a class.**

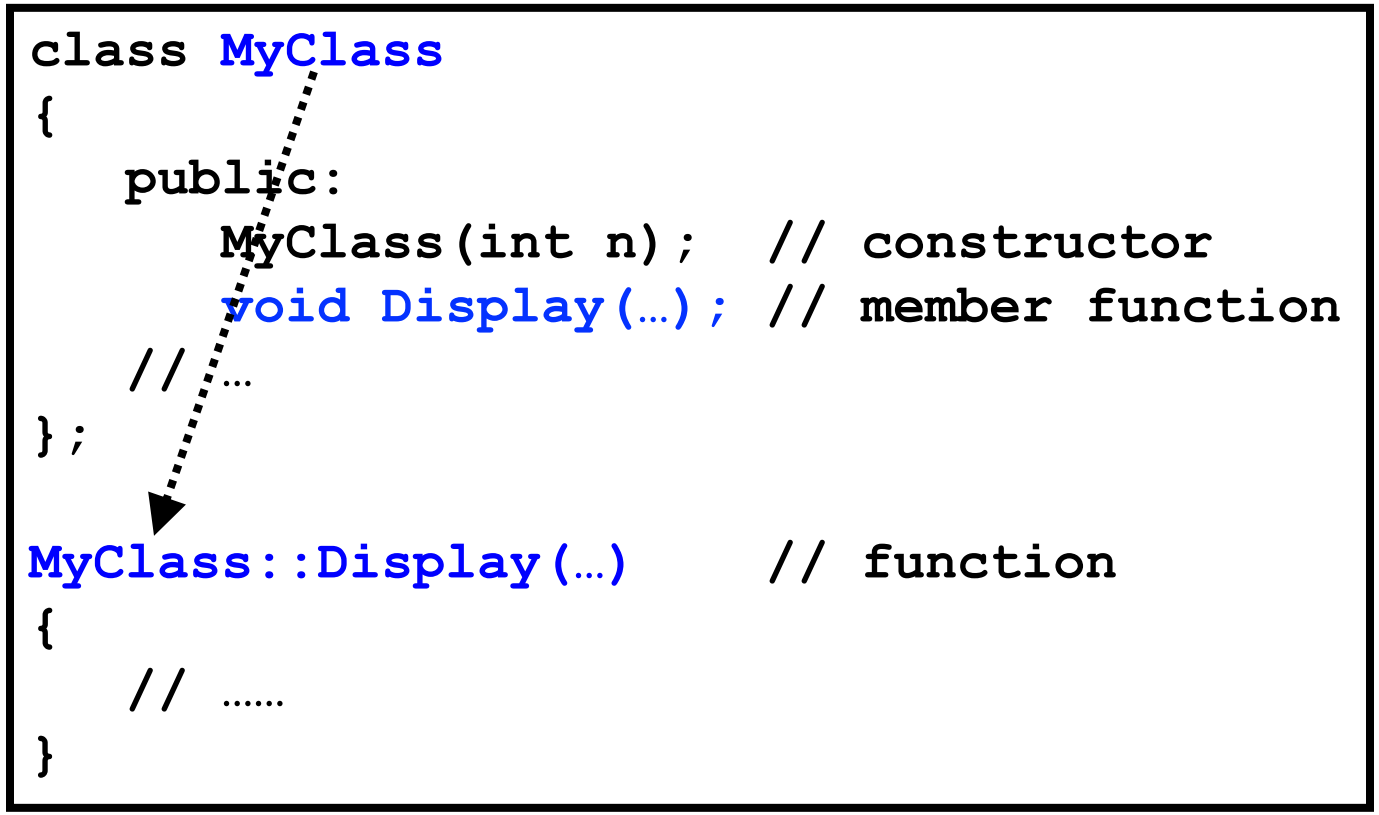


```
class MyClass
{
   public:
      MyClass(int n);   // constructor
   // …
};

MyClass::MyClass(int Input) // constructor function
{
   // …
}
```

```
class Date
{
private:
    int month, day, year;
public:
    //These are constructors
    Date();
    Date(int, int, int);
          . . .
};
Date::Date()
{
    month = 0, day = 0, year = 0;
}
Date::Date(int Month, int Day, int Year)
{
    month = Month;
    day = Day;
    year = Year;
}
```

Member Functions

- Member functions are just functions.



Example

account.cpp

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

class MyAccount
{
   public:
      MyAccount(int Initial_Amount); // constructor
      int  Deposit(int);             // member funct
      int  Withdraw(int);            // member funct
      void Display(void);            // member funct

   private:
      int  Balance;                  // private variable
};
```

account.cpp

```
MyAccount::MyAccount(int initial)
{
   Balance = initial;  // constructor initialization
}

int MyAccount::Deposit(int Amount)
{
   cout << "Deposit Request  = " << Amount << endl;
   cout << "Previous Balance = " << Balance << endl;
   Balance += Amount;
   cout << "New Balance      = " << Balance << endl
        << endl;
   return Balance;
}
```

account.cpp

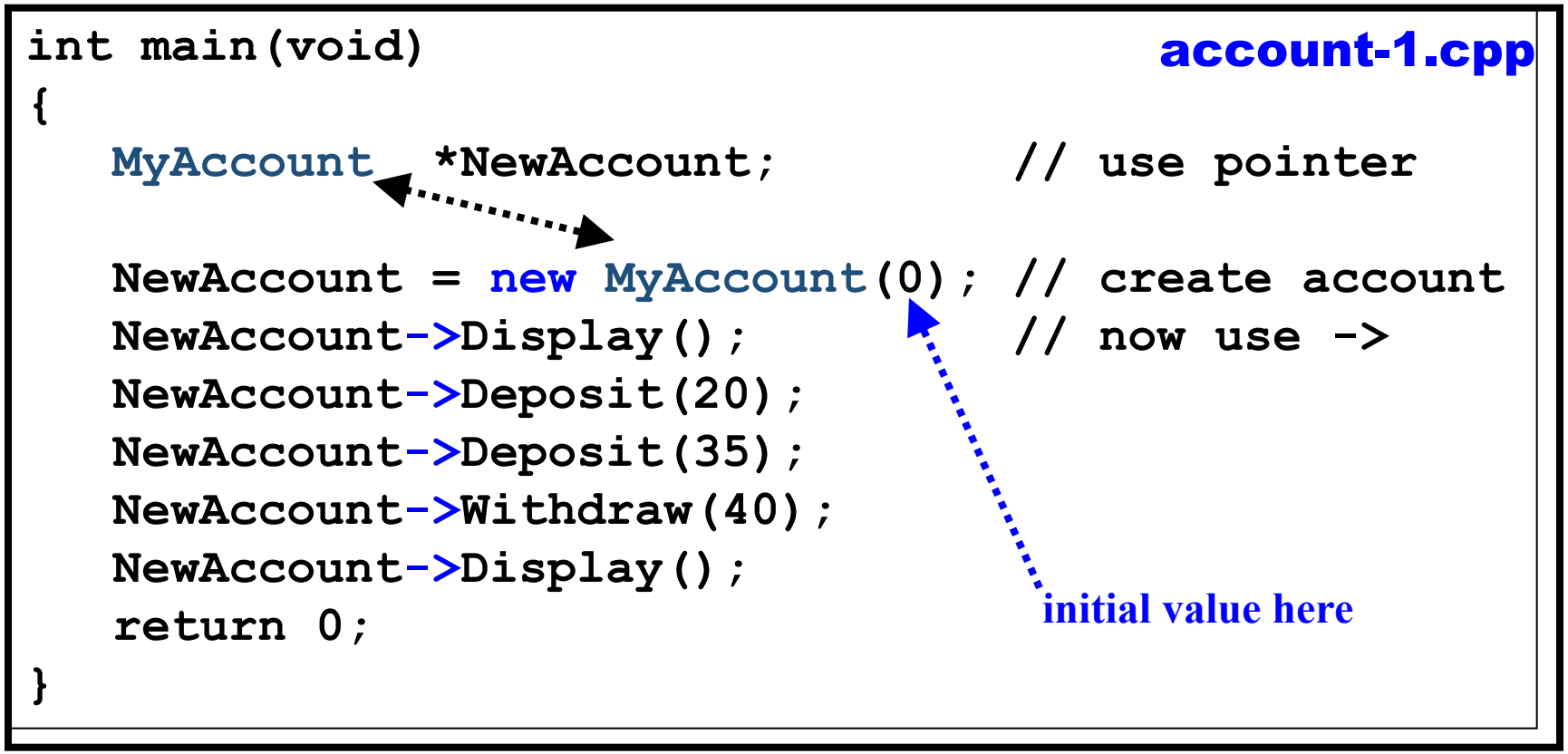
```
int MyAccount::Withdraw(int Amount)
{
   cout << "Withdraw Request = " << Amount << endl;
   cout << "Previous Balance = " << Balance << endl;
   Balance -= Amount;
   cout << "New Balance      = " << Balance << endl
        << endl;
   return Balance;
}

void MyAccount::Display(void)
{
   cout << "Current Balance  = " << Balance << endl
        << endl;
}
```

account.cpp

```
int main(void)
{
   MyAccount  NewAccount(0); // initial new account

   NewAccount.Display();     // display balance
   NewAccount.Deposit(20);   // deposit 20 (Bal=20)
   NewAccount.Deposit(35);   // deposit 35 (Bal=55)
   NewAccount.Withdraw(40);  // withdraw 40 (Bal=15)
   NewAccount.Display();     // current balance
   return 0;
}
```

account-1.cpp

```
int main(void)
{
   MyAccount  *NewAccount;          // use pointer

   NewAccount = new MyAccount(0); // create account
   NewAccount->Display();         // now use ->
   NewAccount->Deposit(20);
   NewAccount->Deposit(35);
   NewAccount->Withdraw(40);
   NewAccount->Display();
   return 0;
}
```



This version uses a pointer.

The `new` operator creates an object and returns a pointer to it.

It is similar to `malloc()` in C. Use `delete` to deallocate.

Constructors : The Initialization Section

- There is a faster way, actually maybe a preferable way, to initialize member variables.

```
class Numbers
{
   public:
      int Lower, Upper;
      Numbers(int a, int b);     // constructor
   // …
};

Numbers::Numbers(int a, int b)
            : Lower(a), Upper(b)   // init. section
{ // function body is empty
}
```

Derived Classes

- Deriving a class from an existing one is called **inheritance** in C++.
- The newly created class is a **derived** class and the class from which the derived class is created is a **base** class.
- The constructor (and destructor) of a base class is not inherited.

- **A derived class is just a class with the following syntax:**

```
class derived-class-name : public base-class-name
{
      public:
          // public member declarations
          derived-class-constructor();
      private:
          // private member declarations
};
```

Derived class inherits everything in the public section of the base class

Small change from original overheads. Available in …/cs3331/cpp/

derived-1.cpp

```
class Base
{
   public:
      int  a;
      Base(int x=10):a(x)
         { cout << "Base constructor" << endl; }
}  // use x to init a
};

class Derived: public Base
{
   public:
      int x;
      Derived(int m=20):x(m) // use m to init x
         { cout << "Derived constructor" << endl; }
};
```

Default value

Base class constructor is called first. Then derived class constructor is called.

Small change from original overheads. Available in …/cs3331/cpp/

derived-1.cpp

```
int main(void)
{
   Base    X, *XX;
   Derived Y, *YY;

   cout << "Base's a value    = " << X.a << endl;
   cout << "Derived's a value = " << Y.a << endl;
   cout << "Derived's x value = " << Y.x << endl;
   cout << endl;
   XX = new Base(123);
   cout << "Base's a value = " << XX->a << endl;
   YY = new Derived(789);
   cout << "Derived's a value = " << YY->a << endl;
   cout << "Derived's x value = " << YY->x << endl;

   return 0;
}
```

derived-2.cpp

```
class Base
{
   public:
      int  a;
      char name[100];
      Base(int);
};

Base::Base(int x = 10)  : a(x)
{
   char  buffer[10];
   strcpy(name, "Class");      // requires string.h
   sprintf(buffer, "%d", a);   // requires stdio.h
   strcat(name, buffer);       // requires string.h
   cout << "Base has " << a << ' ' << name << endl;
}
```

This is not the best way;
but, it works!

derived-2.cpp

```
class Derived: public Base
{
   public:
      Derived(int m=20): Base(m) {   }
};

int main(void)
{
   Base      X(23);
   Derived   Y(789);

   cout << "Base's name    = " << X.name << endl;
   cout << "Derived's name = " << Y.name << endl;

   return 0;
}
```

Call Base constructor with value m

use `m` to call constructor `Base`

"Class23"

"Class789"

Organization & Compilation

- **Normally, the specification part and the implementation part of a class are saved in `.h` and `.cpp` files, respectively.**

User of the class just includes this file

MyAccount.h

```
class MyAccount
{
   public:
      MyAccount(int Initial_Amount);
      int  Deposit(int);
      int  Withdraw(int);
      void Display(void);

   private:
      int  Balance;
};
```

Compile this into object code to link against

MyAccount.cpp

```
#include  <iostream>
#include  "MyAccount.h"

using namespace std;

MyAccount::MyAccount(int initial)
                : Balance(initial)
{ /* function body is empty */ }

int MyAccount::Deposit(int Amount)
{
   cout << "Deposit Request  = " << Amount << endl;
   cout << "Previous Balance = " << Balance << endl;
   Balance += Amount;
   cout << "New Balance      = " << Balance
        << endl << endl;
   return Balance;
}
// other member functions
```

account-3.cpp

```
#include  <iostream>
#include  "MyAccount.h"

using namespace std;

int main(void)
{
   MyAccount  *NewAccount;

   NewAccount = new MyAccount(0);
   NewAccount->Display();
   NewAccount->Deposit(20);
   NewAccount->Deposit(35);
   NewAccount->Withdraw(40);
   NewAccount->Display();
   return 0;
}
```

- **Now we have the specification file `MyAccount.h`, the implementation file `MyAccount.cpp`, and the main program `account-3.cpp`.**
- **Compile the whole thing this way**

```
g++ MyAccount.cpp account-3.cpp -o account-3
```

- **Or, we may compile `MyAccount.cpp` to `MyAccount.o` and use it later:**

```
g++ MyAccount.cpp -c
g++ account-3.cpp MyAccount.o -o account-3
```

The End