Ծրագրավորման Հիմունքներ

Դասախոս՝ Միսակ Սիոյան

Literature

1. C++ How to Program(10th edition, Paul Deitel, Harvey Deitel)
2. Scott Mayers, Effective C++ (3rd edition)
3. Scott Mayers, Effective Modern C++
4. Herb Sutter, Exceptional C++
5. Herb Sutter, More Exceptional C++:
6. Scott Mayers, Effective STL
7. The C++ Programming Language (4th Edition), Bjarne Stroustrup

Introduction to Object Technology

Functions, Member Functions and Classes

- Performing a task in a program requires a **function**.
- In C++, we often create a program unit called a **class** to house the set of functions that perform the class's tasks - these are known as the class's **member functions**.

Instantiation

- You must **build** an **object** from a class before a program can perform the tasks that the class's member functions define.
- The process of doing this is called **instantiation**. An object is then referred to as an **instance** of its class.

```
Class A {…};
A a; // ‘a’ is an instance of class A.
```

Introduction to Object Technology

Class Member Function Calls

- You can send *messages* to an object. Each message is implemented as a **member-function** call that tells a member function of the object to perform its task.

```
Class BankAccount {
…
public:
    void deposit(double val);
…
};
```

```
int main()
{
    BankAccount a;
    a.deposit(1000);

    return 0;
}
```

Introduction to Object Technology

Attributes and Data Members

- An object has **attributes** that it carries along as it's used in a program. These attributes are specified as part of the object's class.
- Attributes are specified by the class's **data members**.

```
Class BankAccount {
…
public:
    void deposit(double val) {
        balance += val;
    }
private:
    double balance;
…
};
```

```
int main()
{
    BankAccount a;
    a.deposit(1000);

    return 0;
}
```

Introduction to Object Technology

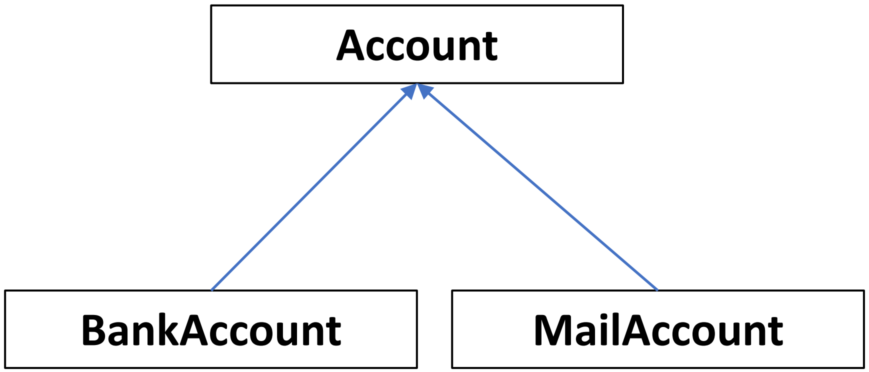
Encapsulation

- Classes **encapsulate** (i.e., wrap) attributes and member functions into objects created from those classes - an object's attributes and member functions are intimately related.
- Objects may communicate with one another, but they're normally not allowed to know how other objects are implemented - implementation details are **hidden** within the objects themselves.
- This information hiding, as we'll see, is crucial to good software engineering.

Introduction to Object Technology

Inheritance

- A new class of objects can be created quickly and conveniently by **inheritance** - the new class absorbs the characteristics of an existing class, possibly customizing them and adding unique characteristics of its own.



Introduction to Object Technology

Polymorphism

- Polymorphism enables you to “program in the **general**” rather than “program in the **specific**”. In particular, you can write programs that process objects of classes that are part of the same class hierarchy as if they were all objects of the hierarchy’s base class:

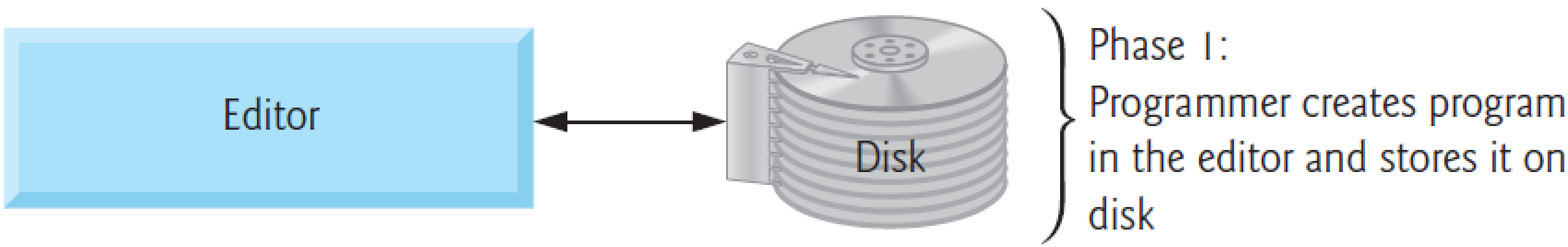
```
// Account is the base class
BankAccount b;
MailAccount m;
FbAccount f;
GoogleAccount g;
Account* accounts[4] = { &a, &m, &f, &g };
for (int i = 0; I < 4; i++) {
    accounts[i]->login();
}
```

C++ Development

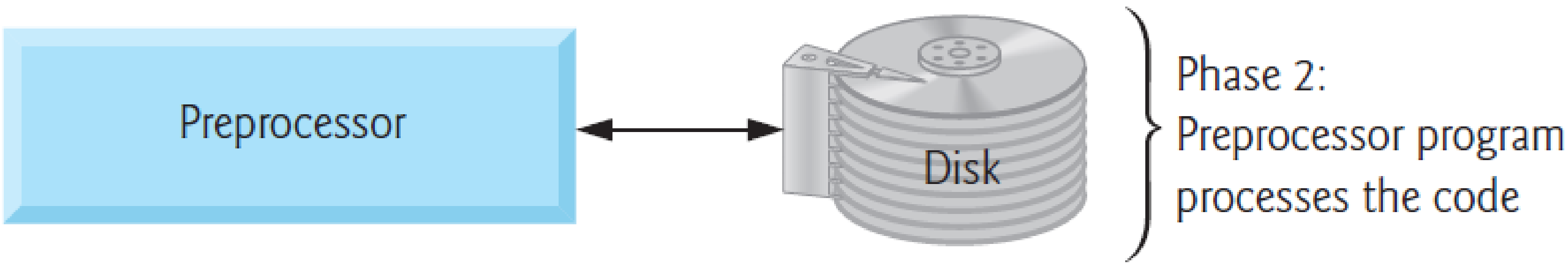
C++ programs typically go through six phases:

1. Edit
2. Preprocess
3. Compile
4. Link
5. Load
6. Execute

C++ Development: Edit



C++ Development: Preprocess



#include <MyClass.h>

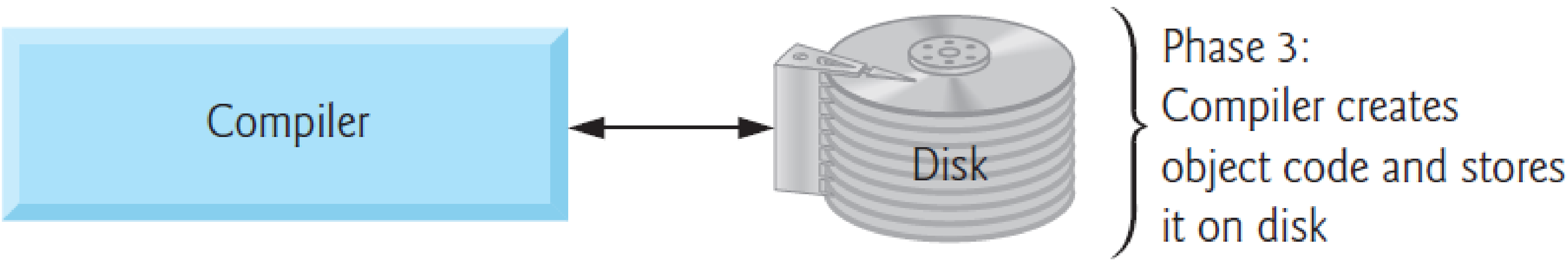
…

#define macroValue 10

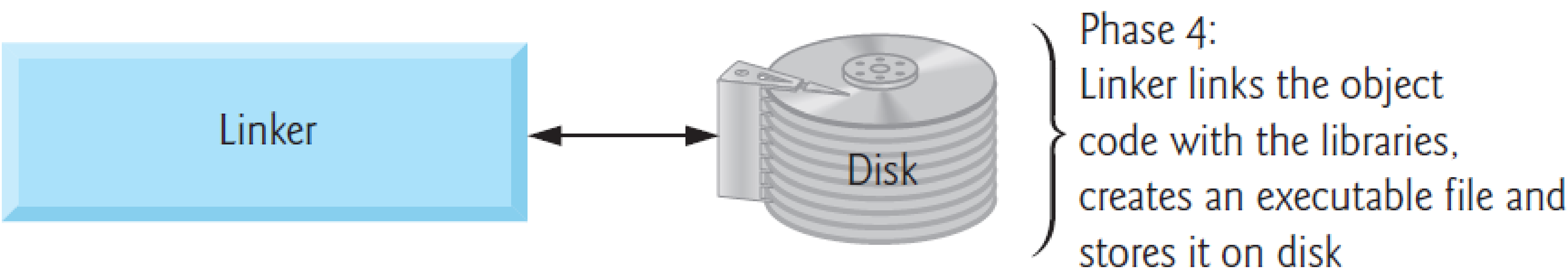
…

int x = macroValue;

C++ Development: Compile

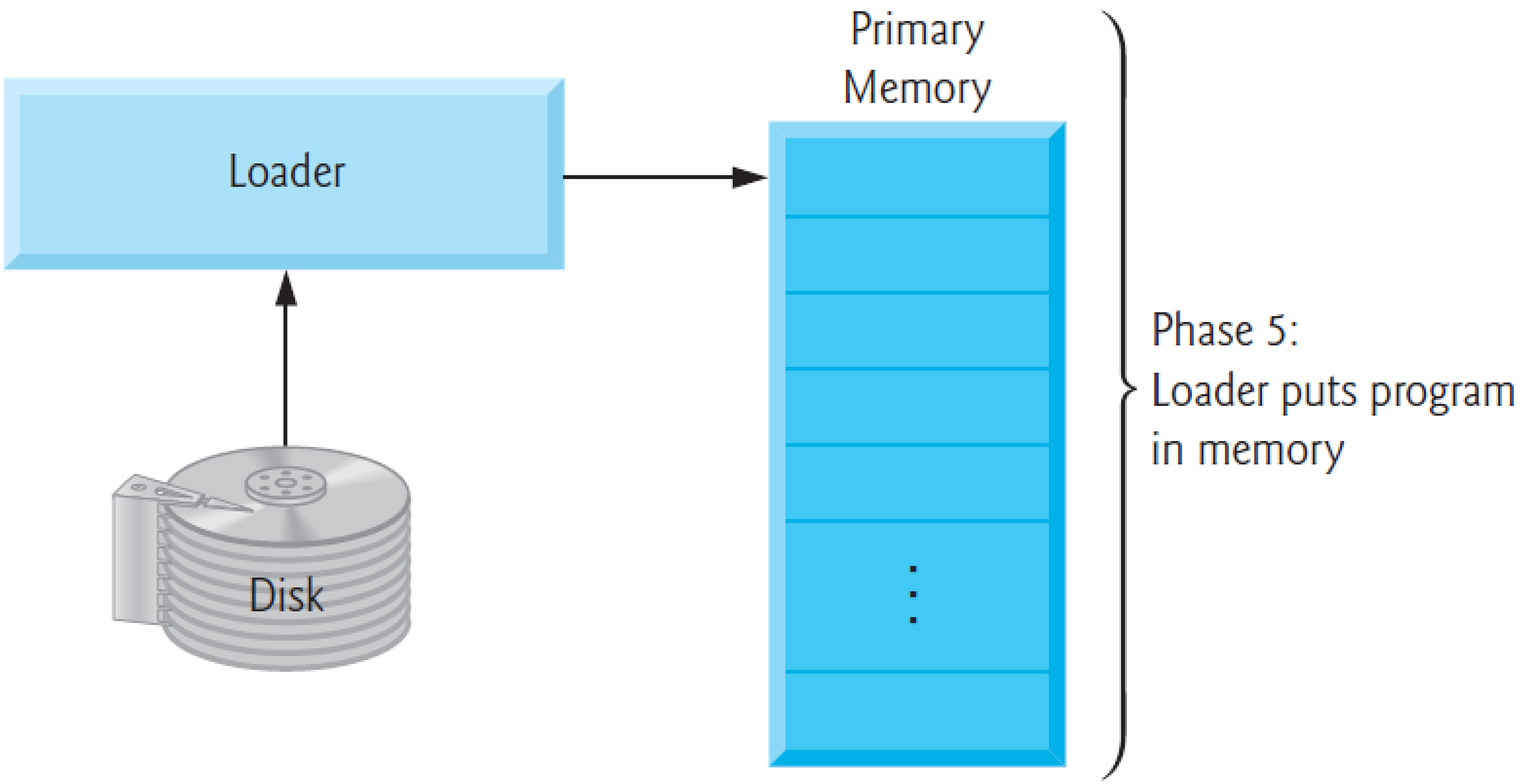


C++ Development: Link

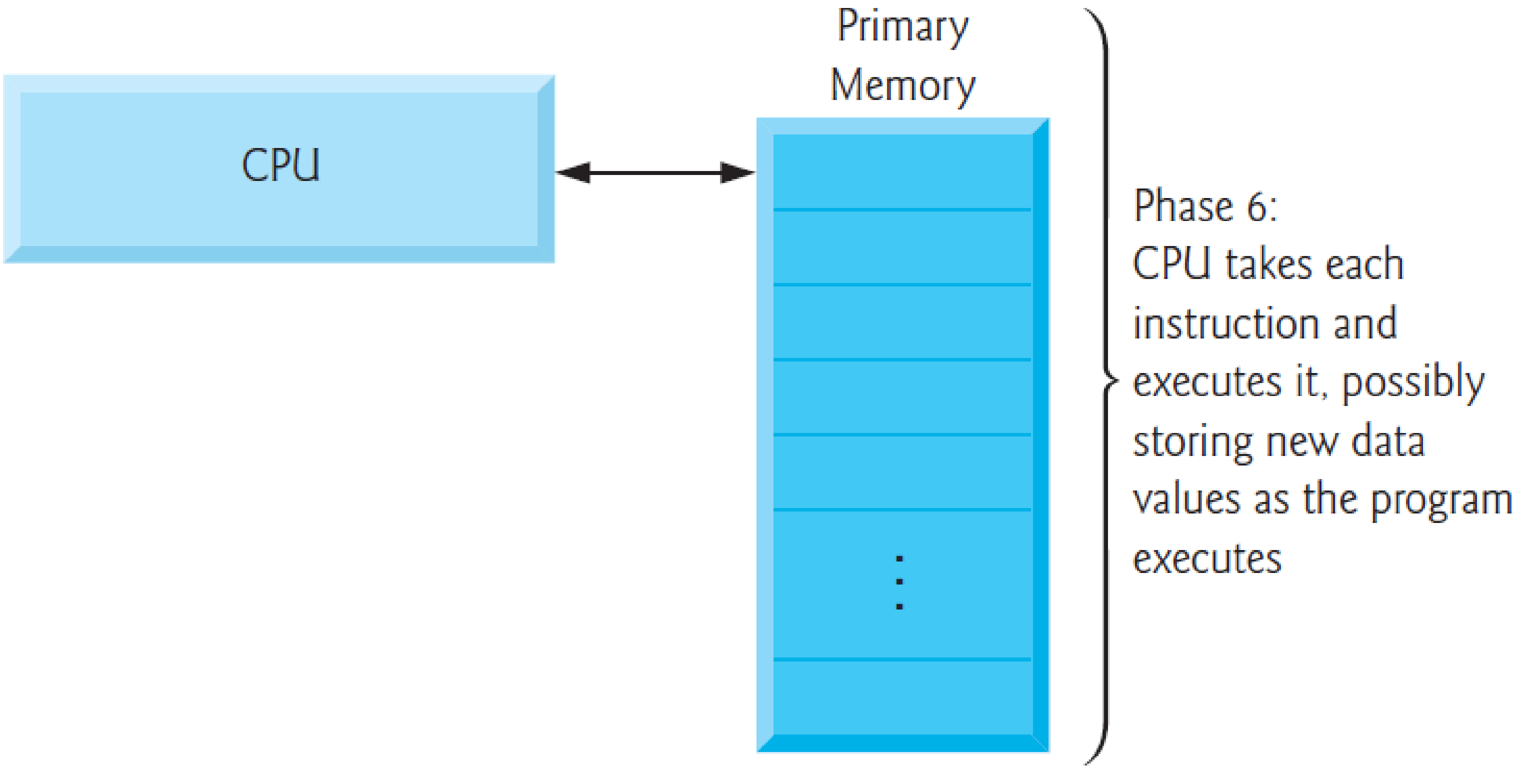


.exe file is created

C++ Development: Load



C++ Development: Execute



Introduction to Classes, Objects and Member Functions

Classes

- Each class you create becomes a **new type** you can use to create objects, so *C++ is an extensible programming language*.
- Classes cannot execute by themselves.

Person object can drive a Car object by telling it what to do (go faster, go slower, turn left, turn right, etc.)—without knowing how the car's internal mechanisms work.
Similarly, the main function can "**drive**" an Account object by calling its member functions—without knowing how the class is implemented. In this sense, main is referred to as a **driver program**.

main.cpp

```
…
Car c;
Person p (&c);
p.driveCar();
```

Account Object

```cpp
// Fig. 3.1: AccountTest.cpp
// Creating and manipulating an Account object.
#include <iostream>
#include <string>
#include "Account.h"

using namespace std;

int main() {
   Account myAccount; // create Account object myAccount

   // show that the initial value of myAccount's name is the empty string
   cout << "Initial account name is: " << myAccount.getName();

   // prompt for and read name
   cout << "\nPlease enter the account name: ";
   string theName;
   getline(cin, theName); // read a line of text
   myAccount.setName(theName); // put theName in myAccount

   // display the name stored in object myAccount
   cout << "Name in object myAccount is: "
      << myAccount.getName() << endl;
}
```

```
Initial account name is:
Please enter the account name: Jane Green
Name in object myAccount is: Jane Green
```

Account Object

Typically, you cannot call a member function of a class until you *create an object* of that class:

```
Account myAccount; // create Account object myAccount
```

The compiler does not know what is **Account:**

```
#include "Account.h“
```

It’s recommended to place a reusable class definition in a file known as a **header** with a **.h** extension: Account.h

Files ending with the **.cpp** filename extension are **source-code** files.

A **string** is actually an object of the *C++ Standard Library* **class string**, which is defined in the header **<string>.**

Account Class definition

```cpp
// Fig. 3.2: Account.h
// Account class that contains a name data member
// and member functions to set and get its value.
#include <string> // enable program to use C++ string data type

class Account {
public:
   // member function that sets the account name in the object
   void setName(std::string accountName) {
      name = accountName; // store the account name
   }

   // member function that retrieves the account name from the object
   std::string getName() const {
      return name; // return name's value to this function's caller
   }
private:
   std::string name; // data member containing account holder's name
}; // end class Account
```

The **camel case** naming is used.

Account Class definition

```
void setName(std::string accountName) {
    name = accountName; // store the name
}
…
myAccount.setName(theName);
```

void – **return type**

accountName – **parameter**

theName – **argument**

{ name = accountName; } – member function **body**

- Parameters are local variables.
- Argument and parameter types must be consistent.

Account Class definition

- By convention, place a class's data members last in the class's body. You can list the class's data members anywhere in the class outside its member-function definitions, but scattering the data members can lead to hard-to-read code.

- Forgetting the semicolon (;) at the end of a class definition is a syntax error!

- Use **std::** with Standard Library Components in Headers. Do not include "using namespace std".

- Declaring a member function with **const** to the **right** of the parameter list tells the compiler, "*this function should not modify the object on which it's called—if it does, please issue a compilation error*." This can help you locate errors if you accidentally insert in the member function code that would modify the object.

```
std::string getName() const {
    return name;
}
```

Access Specifiers

- The **public** and **private** keywords are **access specifiers**.

- The access specifier **private:** indicates that the *data members and member functions are only accessible to* ***class member functions***. This is known as *data hiding* – the members are **encapsulated** (hidden).

- The access specifier **public:** indicates that the *data members and member functions* **are available to the public**.

- By default, everything in a class is **private**, unless you specify otherwise.

- Making a class's data members private and member functions public facilitates debugging because problems with data manipulations are localized to the member functions.

- An attempt by a function that's not a member of a particular class to access a private member of that class is a compilation error.

```
myAccount.name = "name"; // compilation error
```

Initializing Objects with Constructors

- Each class can define a **constructor** that specifies custom initialization for objects of that class:
- A constructor is a special member function that must have the **same name** as the class.
- C++ *requires* a constructor call when each object is created.

```
class Account {
public:
   // constructor initializes data member name with parameter accountName
   explicit Account(std::string accountName)
      : name{accountName} { // member initializer
     // empty body
   }
```

```
Account account1(“Jane Green”);
```

- Normally, constructors are *public*.

Initializing Objects with Constructors

```
class Account {
public:
   // constructor initializes data member name with parameter accountName
   explicit Account(std::string accountName)
      : name{accountName} { // member initializer
     // empty body
   }
```

- The constructor uses a **member-initializer list.**
- *Member initializers* appear between a constructor's parameter list and the left brace that begins the constructor's body.
- The member-initializer list is separated from the parameter list with a colon (:).
- The member initializer list executes **before** the constructor's body executes.
- You can perform initialization in the constructor's body, but it's more ***efficient*** to do it with member initializers, and some types of data members must be initialized this way (will see later).
- Here **explicit** keyword means that the constructor cannot be used by the compiler to perform an *implicit conversion*.

Implicit Conversion

```
class Account {
public:
    Account(int number)
        :_number{number} {
        std::cout << "Account(int number)" << std::endl;
    }

    int _number;
};
void printAccount(Account a) {
    std::cout << "Account number = " << a._number << std::endl;
}
```

```
int main()
{
     Account a{12};
     printAccount(a);
     printAccount(789);
     return 0;
}
```

```
…
Account(int number)
Account number = 12
Account(int number)
Account number = 789
```

Implicit Conversion: explicit keyword

```
class Account {
public:
    explicit Account(int number)
        :_number{number} {
        std::cout << "Account(int number)" << std::endl;
    }

    int _number;
};
void printAccount(Account a) {
    std::cout << "Account number = " << a._number << std::endl;
}

int main()
{
    Account a{12};
    printAccount(a);
    printAccount(789); // error: Could not convert ‘789’ from ‘int’ to ‘Account’
    return 0;
}
```

Instantiating Account Objects

- The constructor uses a **member-initializer list.**

```
// Fig. 3.5: AccountTest.cpp
// Using the Account constructor to initialize the name data
// member at the time each Account object is created.
#include <iostream>
#include "Account.h"

using namespace std;

int main() {
   // create two Account objects
   Account account1{"Jane Green"};
   Account account2{"John Blue"};

   // display initial value of name for each Account
   cout << "account1 name is: " << account1.getName() << endl;
   cout << "account2 name is: " << account2.getName() << endl;
}
```

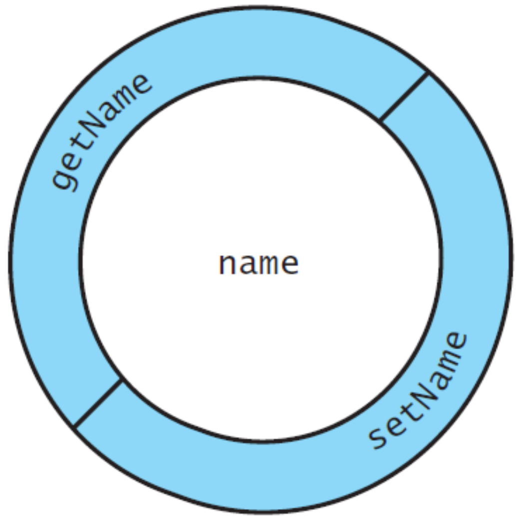
```
account1 name is: Jane Green
account2 name is: John Blue
```

Default Constructor

```
Account myAccount;
```

- Here C++ *implicitly* calls the class's **default constructor.**
- In any class that does not explicitly define a constructor, the compiler provides a default constructor with no parameters.
- The default constructor does not initialize the class's fundamental-type data members *but does call the default constructor for each data member that's an object of another class*.
- An uninitialized fundamental-type variable contains an undefined ("garbage") value.
- There's *no default constructor* in a class that defines a constructor.
- *Unless default initialization of your class's data members is acceptable*, you should generally provide a custom constructor to ensure that your data members are properly initialized with meaningful values when each new object of your class is created.

Encapsulation



Conceptual view of an Account object with its encapsulated private data member name and protective layer of public member functions

Encapsulation

- Any client code that needs to interact with the Account object can do so *only* by calling the public **set** and **get** member functions.

- Generally, *data members* should be **private** and *member functions* **public**. Later you will see why you might use a public data member or a private member function.

- Using *public* set and get functions to control access to *private* data makes programs **clearer and easier to maintain**. Change is the rule rather than the exception. **You should anticipate that your code will be modified, and possibly often**.

List Initialization (C++ 11)

`Before C++ 11`

int number1 = **0**; // first integer to add (initialized to 0)
int number2 = **0**; // second integer to add (initialized to 0)
int sum = **0**; // sum of number1 and number2 (initialized to 0)

After C++ 11 (List Initialization or brace initialization)

int number1{**0**}; // first integer to add (initialized to 0)
int number2{**0**}; // second integer to add (initialized to 0)
int sum{**0**}; // sum of number1 and number2 (initialized to 0)

Examples:

Int number1(0);
Int number2{0};
Int number3 = 0;
int number4 = {0},

- *Declare only one variable in each declaration* and provide a **comment** that explains the variable's purpose in the program.

List Initialization (C++ 11)

- For *fundamental-type* variables, list-initialization syntax **prevents narrowing conversions** that could result in data loss.

```
int x = 12.7; // 0.7 is truncated
int x = {12.7}; // Error!
```

Error: *Type 'double' cannot be narrowed to 'int' in initializer list.*

```
short s = 32768; // Range of short: [-32768 to 32767]
short s = {32768}; // Error!
```

Error: *Constant expression evaluates to 32768 which cannot be narrowed to type 'short'.*

Account Class

```cpp
// Fig. 3.8: Account.h
// Account class with name and balance data members, and a
// constructor and deposit function that each perform validation.
#include <string>

class Account {
public:
   // Account constructor with two parameters
   Account(std::string accountName, int initialBalance)
      : name{accountName} { // assign accountName to data member name

      // validate that the initialBalance is greater than 0; if not,
      // data member balance keeps its default initial value of 0
      if (initialBalance > 0) { // if the initialBalance is valid
         balance = initialBalance; // assign it to data member balance
      }
   }

   // function that deposits (adds) only a valid amount to the balance
   void deposit(int depositAmount) {
      if (depositAmount > 0) { // if the depositAmount is valid
         balance = balance + depositAmount; // add it to the balance
      }
   }
```

Account Class

```
    // function returns the account balance
    int getBalance() const {
       return balance;
    }

    // function that sets the name
    void setName(std::string accountName) {
       name = accountName;
    }

    // function that returns the name
    std::string getName() const {
       return name;
    }
 private:
    std::string name; // account name data member
    int balance{0}; // data member with default initial value
 }; // end class Account
```

- **int** balance{**0**}; declares a data member balance of type int and initializes its value to 0. This is known as an **in-class initializer** and was introduced in **C++11.** For C++<11 versions it will give a compile error.
- Every object of class Account contains its **own** copies of **both** the name and the balance.

Account Class

```cpp
// Fig. 3.9: AccountTest.cpp
// Displaying and updating Account balances.
#include <iostream>
#include "Account.h"

using namespace std;

int main()
{
   Account account1{"Jane Green", 50};
   Account account2{"John Blue", -7};

   // display initial balance of each object
   cout << "account1: " << account1.getName() << " balance is $"
      << account1.getBalance();
   cout << "\naccount2: " << account2.getName() << " balance is $"
      << account2.getBalance();

   cout << "\n\nEnter deposit amount for account1: "; // prompt
   int depositAmount;
   cin >> depositAmount; // obtain user input
   cout << "adding " << depositAmount << " to account1 balance";
   account1.deposit(depositAmount); // add to account1's balance

   // display balances
   cout << "\n\naccount1: " << account1.getName() << " balance is $"
      << account1.getBalance();
   cout << "\naccount2: " << account2.getName() << " balance is $"
      << account2.getBalance();

```

Account Class

```
   cout << "\n\nEnter deposit amount for account2: "; // prompt
   cin >> depositAmount; // obtain user input
   cout << "adding " << depositAmount << " to account2 balance";
   account2.deposit(depositAmount); // add to account2 balance

   // display balances
   cout << "\n\naccount1: " << account1.getName() << " balance is $"
      << account1.getBalance();
   cout << "\naccount2: " << account2.getName() << " balance is $"
      << account2.getBalance() << endl;
}
```

```
account1: Jane Green balance is $50
account2: John Blue balance is $0

Enter deposit amount for account1: 25
adding 25 to account1 balance

account1: Jane Green balance is $75
account2: John Blue balance is $0

Enter deposit amount for account2: 123
adding 123 to account2 balance

account1: Jane Green balance is $75
account2: John Blue balance is $123
```

Account Class

- Replacing **duplicated** code with calls to a function that contains *only one* copy of that code can reduce the size of your program and improve its maintainability.

- Most C++ compilers issue a *warning* if you attempt to use the value of an *uninitialized* variable. This helps you avoid dangerous execution-time logic errors. It's always better to get the warnings and errors out of your programs at compilation time rather than execution time.

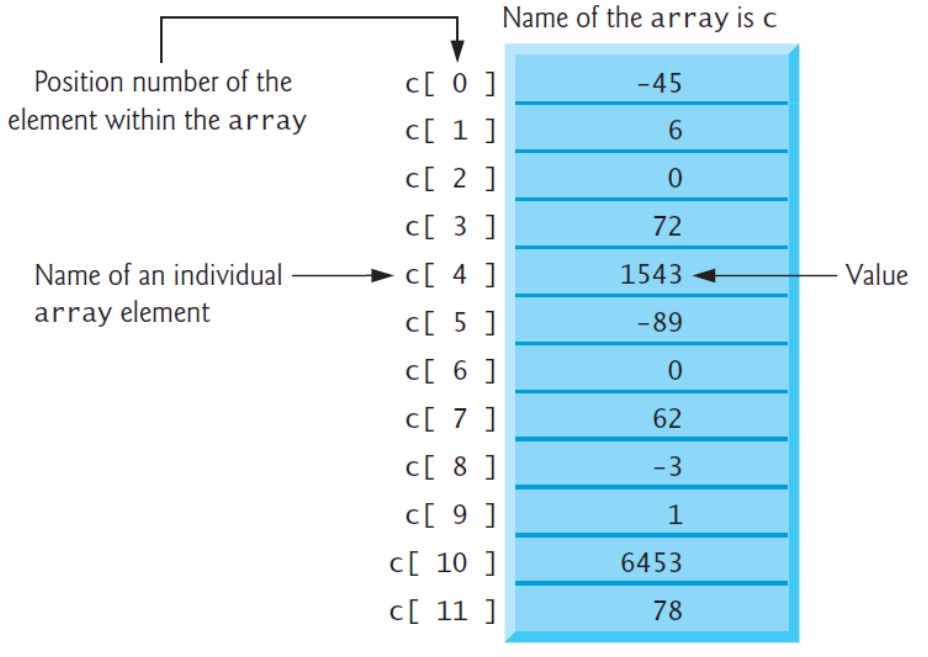
```
…
int a;
std::cout << a;
…
```

Compiler Warning: **1st function call argument is an uninitialized value**.

Class Template **std::array** **[C++ 11]**

Arrays

- Arrays are *fixed-size sequence of elements of a* ***given type*** *where the number of elements is specified at compile time*.
- There are two types of fixed-size arrays in C++
  - C-style arrays or built-in arrays
  - std::array



C-style array

C-style array is declared as:

type **arrayName** [*arraySize*];

The compiler reserves the appropriate amount of memory at **compile time** and allocates the array in **stack**. The *arraySize* must be an **integer constant** greater than zero.

For example, to tell the compiler to reserve 12 elements for integer array c, use the declaration

int c[12]; // c is an array of 12 integers

C-style array

```
// Fig. 7.3: fig07_03.cpp
// Initializing an array.
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

int main()
{
  int n[ 10 ]; // n is an array of 10 integers

  // initialize elements of array n to 0
  for ( int i = 0; i < 10; i++ )
    n[ i ] = 0; // set element at location i to 0

  cout << "Element" << setw( 13 ) << "Value" << endl;

  // output each array element's value
  for ( int j = 0; j < 10; j++ )
    cout << setw( 7 ) << j << setw( 13 ) << n[ j ] << endl;

  return 0; // indicates successful termination
} // end main
```

```
Element        Value
      0            0
      1            0
      2            0
      3            0
      4            0
      5            0
      6            0
      7            0
      8            0
      9            0
```

std::array

std::array is a **class template** declared as:

type **arrayName** [*arraySize*];

std::array<*type*, *arraySize*> **arrayName**;

The compiler reserves the appropriate amount of memory at **compile time** and allocates the array in **stack**. The *arraySize* must be an **integer constant** greater than zero.

For example, to tell the compiler to reserve 12 elements for integer array c, use the declaration

std::array<int, 12> c; // c is an array of 12 int values

std::array

```
// Fig. 7.3: fig07_03.cpp
// Initializing an array's elements to zeros and printing the array.
#include <iostream>
#include <iomanip>
#include <array>
using namespace std;

int main() {
   array<int, 5> n; // n is an array of 5 int values

   // initialize elements of array n to 0
   for (size_t i{0}; i < n.size(); ++i) {
      n[i] = 0; // set element at location i to 0
   }

   cout << "Element" << setw(10) << "Value" << endl;

   // output each array element's value
   for (size_t j{0}; j < n.size(); ++j) {
      cout << setw(7) << j << setw(10) << n[j] << endl;
   }
}
```

```
Element     Value
      0         0
      1         0
      2         0
      3         0
      4         0
```

std::array

- According to the C++ standard **size_t** represents an unsigned integral type.
- Type **size_t** is defined in the **std** namespace and is in header <**cstddef**>.
- If there are *fewer* initializers than array elements, the remaining array elements are initialized to zero

```
array<int, 5> n{}; // initialize elements of array n to 0
```

- The number of initializers *must be less than or equal to the array size*. This array declaration causes a **compilation error**

```
array<int, 5> n{32, 27, 64, 18, 95, 14};
```

std::array with initializer list

```cpp
// Fig. 7.4: fig07_04.cpp
// Initializing an array in a declaration.
#include <iostream>
#include <iomanip>
#include <array>
using namespace std;

int main() {
   array<int, 5> n{32, 27, 64, 18, 95}; // list initializer

   cout << "Element" << setw(10) << "Value" << endl;

   // output each array element's value
   for (size_t i{0}; i < n.size(); ++i) {
      cout << setw(7) << i << setw(10) << n[i] << endl;
   }
}
```

```
Element     Value
      0        32
      1        27
      2        64
      3        18
      4        95
```

std::array vs C-style array

- An array is best understood as a built-in array with its **size** firmly attached, without implicit, potentially surprising conversions to pointer types, and with a few convenience functions provided.
- std::array has a lot of functionality (**arrays can be assigned to each other**):

array::at	C++11
array::back	C++11
array::begin	C++11
array::cbegin	C++11
array::cend	C++11
array::crbegin	C++11
array::crend	C++11
array::data	C++11
array::empty	C++11

array::empty	C++11
array::end	C++11
array::fill	C++11
array::front	C++11
array::max_size	C++11
array::operator[]	C++11
array::rbegin	C++11
array::rend	C++11
array::size	C++11
array::swap	C++11

std::array vs C-style array

- std::arrays can be assigned to each other:

```
const size_t size = 4;
std::array<int, size> arr1 {1,2,3,4};
std::array<int, size> arr2 {7,8,9,10};
arr1 = arr2;
for(size_t i{0}; i < arr1.size(); ++i) {
    std::cout << arr1[i] << " ";
}
```

Result: **7 8 9 10**

How std::array is implemented?

Consider this class:

```
class Arr {
public:
    int _arr[5];
};

int main()
{
    Arr ar = {1, 2, 3, 4, 5};
    for(size_t i{0}; i < 5; ++i) {
        std::cout << ar._arr[i] << " ";
    }
}
```

Result: **1 2 3 4 5**

How std::array is implemented?

The std::array is implemented like this:

```
template< class T, std::size_t N>
struct arrayNew {
    size_t size();
    T& operator[](size_t idx);
    T _data[N];
    // Plus a lot of stuff.
};

template <class T, std::size_t N>
size_t arrayNew<T,N>::size() {
    return N;
}
template <class T, std::size_t N>
T& arrayNew<T,N>::operator[](size_t idx) {
    return _data[idx];
}
```

How std::array is implemented?

The std::array is implemented like this:

```
arrayNew<int, 5> arrNew {5,55,555,5555,55555};
arrNew[2] = 2000;
for(size_t i{0}; i < arrNew.size(); ++i) {
    std::cout << arrNew[i] << " ";
}
std::cout << std::endl;
```

Result: **5 55 2000 5555 55555**

Range-based for statement [C++ 11]

- The C++11 range-based for statement allows you to do iterate **without using a counter.**

- It avoids the possibility of "**stepping outside**" the array and *eliminates the need for you to implement your own bounds checking*.

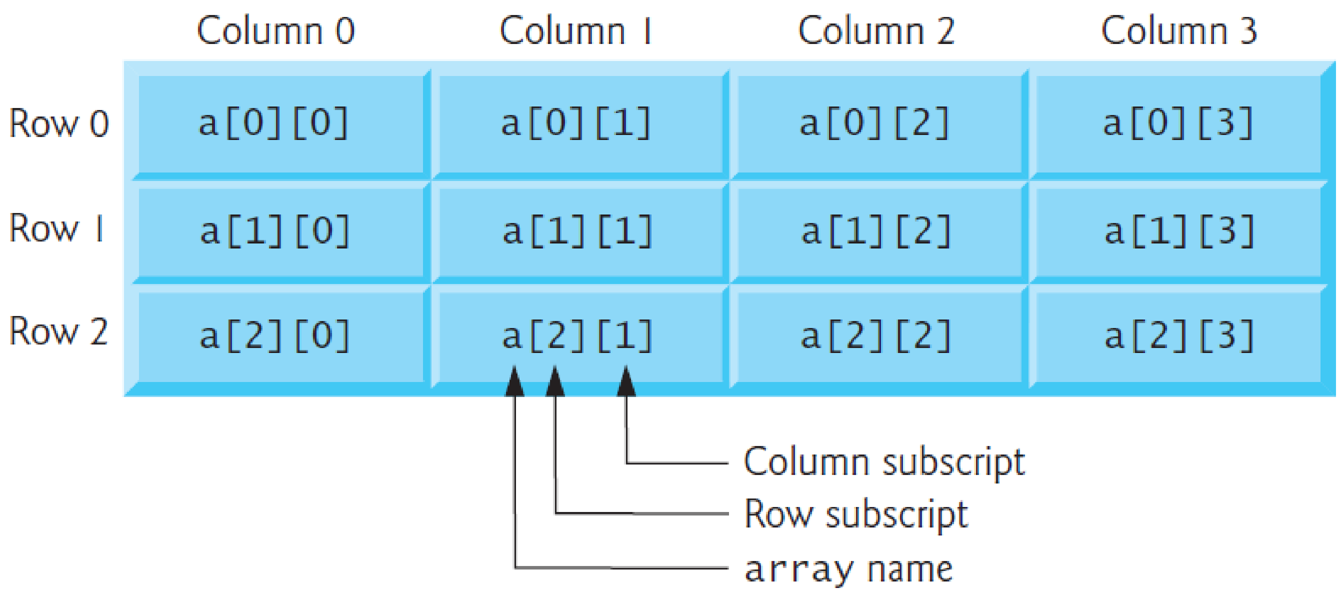
for (*type* item : container) {
}

- It assumes that **begin()** and **end()** functions are implemented for container.

```cpp
// Fig. 7.11: fig07_11.cpp
// Using range-based for to multiply an array's elements by 2.
#include <iostream>
#include <array>
using namespace std;

int main() {
   array<int, 5> items{1, 2, 3, 4, 5};

   // display items before modification
   cout << "items before modification: ";
   for (int item : items) {
      cout << item << " ";
   }

   // multiply the elements of items by 2
   for (int& itemRef : items) {
      itemRef *= 2;
   }

   // display items after modification
   cout << "\nitems after modification: ";
   for (int item : items) {
      cout << item << " ";
   }

   cout << endl;
}
```

```
items before modification: 1 2 3 4 5
items after modification: 2 4 6 8 10
```

Multidimensional arrays



- You can use arrays with two dimensions (i.e., subscripts) to represent tables of values consisting of information arranged in **rows** and **columns**.
- Referencing a two-dimensional array element **a[x][y]** incorrectly as **a[x, y]** *is an error*. Actually, a[x, y] is treated as a[y], because C++ evaluates the expression x, y (containing a comma operator) simply as y (the last of the comma-separated expressions).

Multidimensional arrays example

```
// Fig. 7.17: fig07_17.cpp
// Initializing multidimensional arrays.
#include <iostream>
#include <array>
using namespace std;

const size_t rows{2};
const size_t columns{3};
void printArray(const array<array<int, columns>, rows>&);

```

Multidimensional arrays example

```
int main() {
   array<array<int, columns>, rows> array1{1, 2, 3, 4, 5, 6};
   array<array<int, columns>, rows> array2{1, 2, 3, 4, 5};

   cout << "Values in array1 by row are:" << endl;
   printArray(array1);

   cout << "\nValues in array2 by row are:" << endl;
   printArray(array2);
}

// output array with two rows and three columns
void printArray(const array<array<int, columns>, rows>& a) {
   // loop through array's rows
   for (auto const& row : a) {
      // loop through columns of current row
      for (auto const& element : row) {
         cout << element << ' ';
      }

      cout << endl; // start new line of output
   }
}
```

```
Values in array1 by row are:
1 2 3
4 5 6

Values in array2 by row are:
1 2 3
4 5 0
```

auto keyword [C++ 11]

- **auto** keyword tells the compiler to infer (determine) a variable's data type based on the variable's *initializer value*

```
auto i = 10;       // int
auto b = true;   // bool
auto d = 12.3;  // double
```