Pointers and Dynamic Memory Management

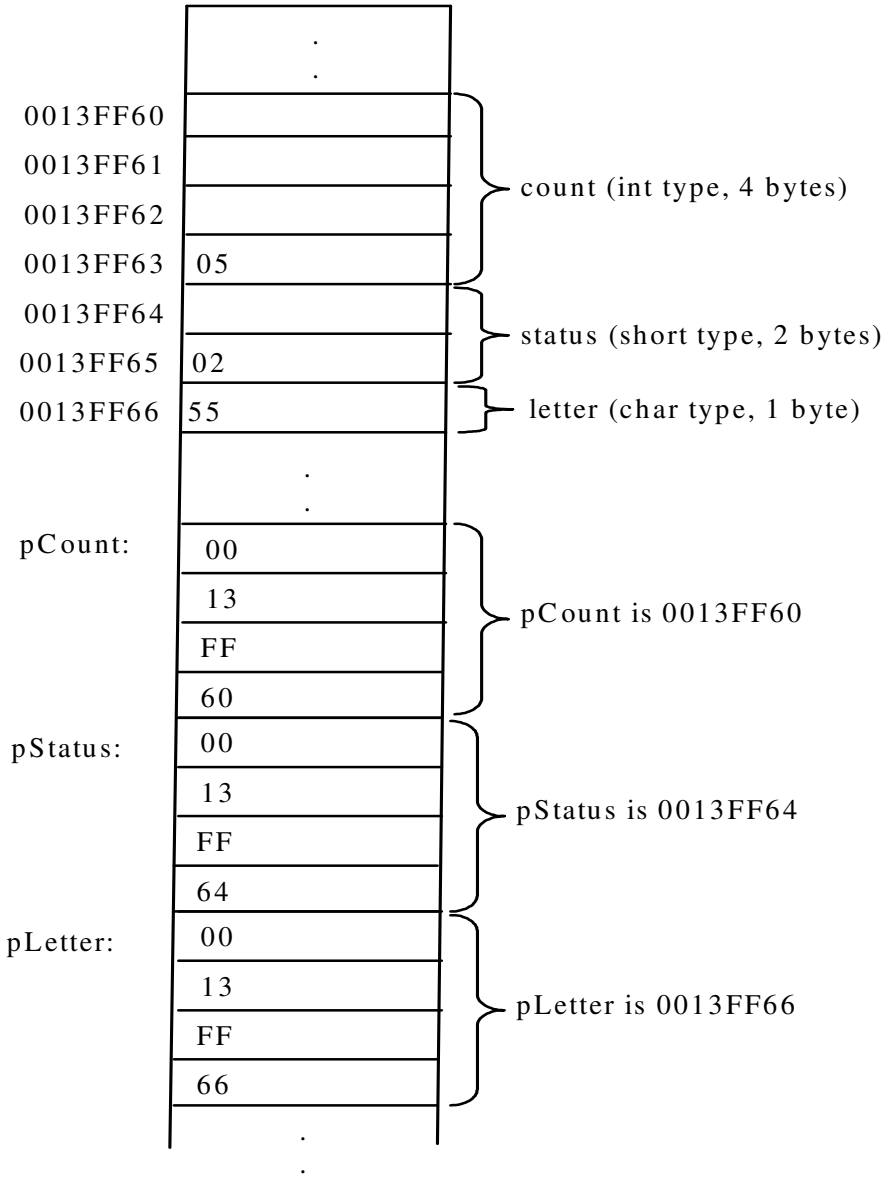
Objectives

- To describe what a pointer is.
- To learn how to declare a pointer and assign a memory address to it.
- To access values via pointers.
- To define synonymous types using the **typedef** keyword.
- To declare constant pointers and constant data.
- To explore the relationship between arrays and pointers.
- To access array elements using pointers.
- To pass pointer arguments to a function.
- To learn how to return a pointer from a function.
- To use the **new** operator to create dynamic arrays.
- To create objects dynamically and access objects via pointers.
- To use smart pointers to avoid memory leak and dangling references .
- To reference the calling object using the **this** pointer.
- To implement the destructor for performing customized operations.
- To design a class for students registering courses.
- To create an object using the copy constructor that copies data from another object of the same type.
- To customize the copy constructor for performing a deep copy.

What is a Pointer?

Pointer variables, simply called *pointers*, are designed to hold memory addresses as their values. Normally, a variable contains a specific value, e.g., an integer, a floating-point value, and a character. However, a pointer contains the memory address of a variable that in turn contains a specific value.

What is a Pointer?




```
int count = 5;
short status = 2;
char letter = 'A';

int* pCount = &count;
short* pStatus = &status;
char* pLetter = &letter;

pCount = &count;
```

Declare a Pointer

Like any other variables, pointers must be declared before they can be used. To declare a pointer, use the following syntax:

dataType* pVarName;

Each variable being declared as a pointer must be preceded by an asterisk (*). For example, the following statement declares a pointer variable named pCount that can point to an int varaible.

int* pCount;

TestPointer

Dereferencing

Referencing a value through a pointer is called *indirection*. The syntax for referencing a value from a pointer is

*pointer

For example, you can increase count using

count++; // direct reference, increment the value in count by 1

or

(*pCount)++; // indirect reference, the value in the memory pointed by pCount is incremented by 1

Pointer Type

A pointer variable is declared with a type such as int, double, etc. You have to assign the address of the variable of the same type. It is a syntax error if the type of the variable does not match the type of the pointer. For example, the following code is wrong.

```
int area = 1;
double* pArea = &area; // Wrong
```

Initializing Pointer

Like a local variable, a local pointer is assigned an arbitrary value if you don't initialize it. A pointer may be initialized to nullptr, which is a special value for a pointer to indicate that the pointer points to nothing. You should always initialize pointers to prevent errors. Dereferencing a pointer that is not initialized could cause fatal runtime error or it could accidentally modify important data.

```
int* p1 = nullptr;

int* p2 = NULL;

int* p3 = 0;
```

Caution

You can declare two variables on the same line. For example, the following line declares two int variables:

```
int i = 0, j = 1;
```

Can you declare two pointer variables on the same line as follows?

```
int* pI, pJ;
```

No, this line is equivalent to

```
int *pI, pJ;
```

typedef

A synonymous type can be defined using the **typedef** keyword.

Syntax:

typedef existingType newType;

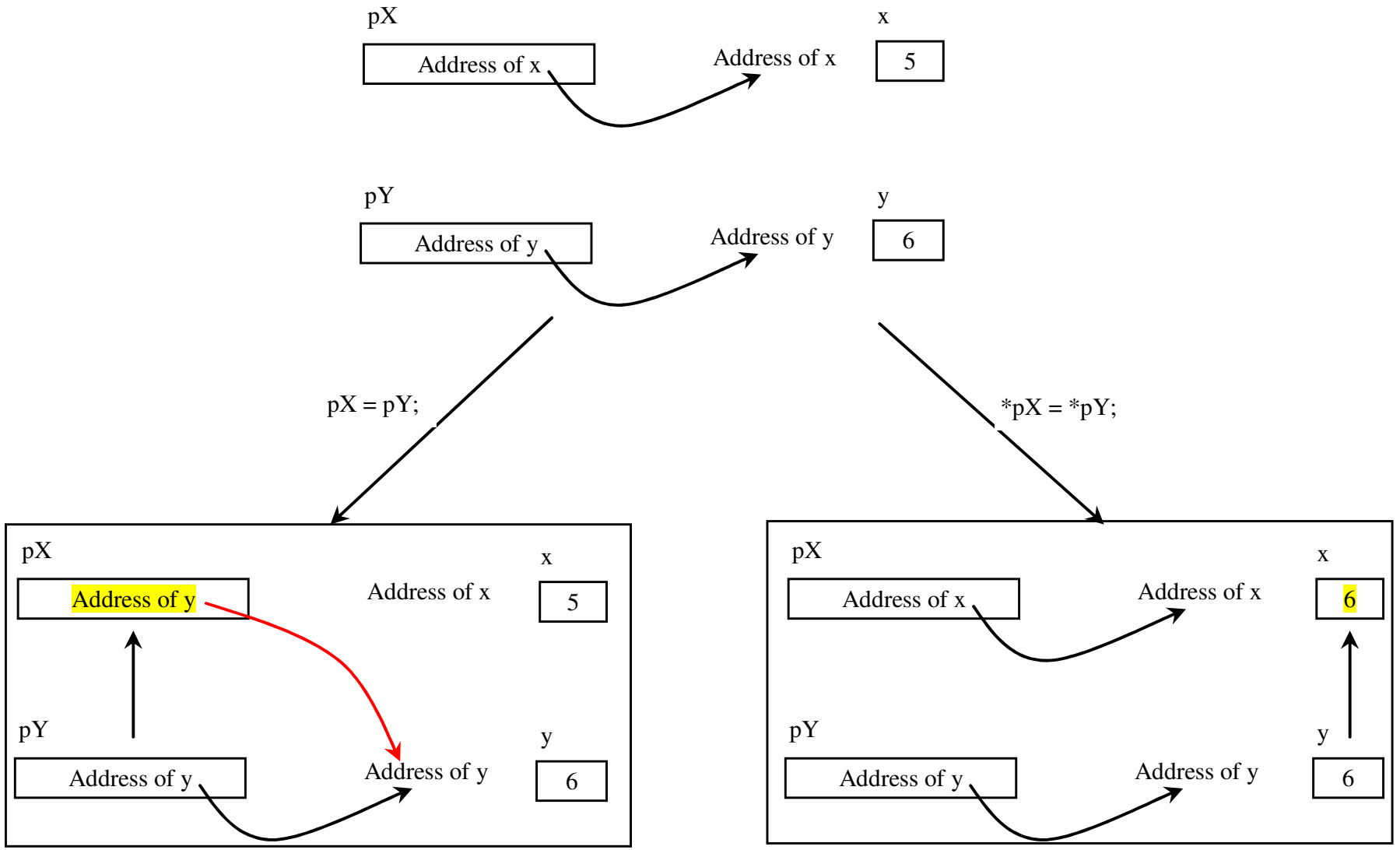
Example:
typedef int integer;
typedef int* intPointer;

integer value = 40;
intPointer p1, p2; // This declares two pointers of int type.

Effect of Assignment =

Suppose pX and pY are two pointer variables for variables x and y. To understand the relationships between the variables and their pointers, let us examine the effect of assigning pY to pX and *pY to *pX.

Effect of Assignment =

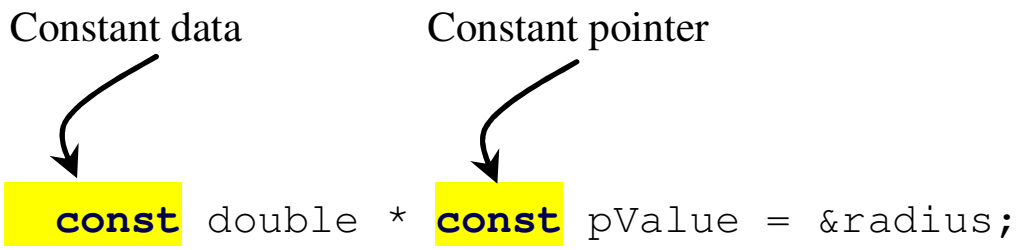


Using const with Pointers

You learned how to declare a constant using the const keyword. A constant cannot be changed once it is declared. You can declare a constant pointer. For example, see the following code:

double radius = 5;

double* const pValue = &radius;



Using const with Pointers

Rule 1: A const variable must always be initialized

```
const int h; // Wrong
```

Rule 2: The address of a constant variable can only be assigned to a "pointer to a constant variable".

```
const float a=3.60;// OK, based on Rule 1
```

```
float *p=&a; // wrong,
```
pointer p is not a pointer to a constant.

```
const double *pc;
pc=&a; // OK, a is a constant and pc is a pointer
       // to a constant variable!
```

Using const with Pointers

Rule 3:

The value of the object "pointer to a constant" (pc in this case) points to can not be changed via the pointer itself.

```
*pc = 3.14   //Wrong, the type of the object pc
              //points to is const.
```

Rule 4:

A "pointer to a constant" need not be initialized at declaration time. It may be initiated later by a constant or non-constant variable.

Using const with Pointers

Rule 5:

Pointer to a constant variable can be changed to point to another variable of the same type.(e.g. to contain the address of another (non-constant) variable).

```
float d;
pc = &d; // OK, since pc isn't a constant pointer!
```

This is a missleading operation. Since when we declared pc we wanted it to point to a constant variable.

Nayeb.Maleki@miun.se

Using const with Pointers

Rule 6:

” const int* " **is the same as** ” int const* "

Example:

`const int* p;` is the same as `int const* p;`

NOTE!

"pointers to constants" are often used as formal arguments to functions, when you do not want to change the content of the argument inside the function. See next slide for example:

Nayeb.Maleki@miun.se

Using const with Pointers : example

```
#include <iostream>
using namespace std;
void function(const int* array);

int main()
{
    int array[10];
    for (int i=0; i< 10; i++)
        array[i]=i;
    function(array);
    return 0;
} //end-main()

void function(const int* array) {
    *(array + 3) = 2; //WRONG
}
```

Using const pointers :

Rule 7:

As for an ordinary constant variable, a pointer that is constant must be initialized when you declare it.

```
int * const panda;  //WRONG,Must be initialized
int aInt , anotherInt;
int *const pek = &aInt;
//OK ,pek is a constant pointer to an int.
```

Rule 8:

One can modify (change) the value of the object, which the constant pointer points to.

```
*pek = 12;  // ok, since aInt isn't a const
```

Using const pointers :

Rule 9:

You can not modify (change) the value (it is an address) of a constant pointer (That is one can not change the direction of the constant pointer).

```
pek = &anotherInt;  //WRONG
```

Rule 10:

You can not convert from " `const int*` " or

" `int const *` " (these were the same thing) to

" `int * const` ". See next slide :

Nayeb.Maleki@miun.se

Using const pointers : example

```
const int *c;
```
(pointer to a `const int`) // OK

```
int * const p = c;
```
(constant pointer to a int) //WRONG

IMPORTANT!:

p and *c both are constants but *p och c are NOT.

In the case of *c, we assume that the pointer points to a constant ofcourse!

Nayeb.Maleki@miun.se

Constant pointer to a Constant object

Rule 11:

Must be initialized.

```
const int p = 1;      //A constant variable
const int *const pc = &p; // constanta pointer
                          //must be initialized
```

pc is a constant pointer to a constant object.

Rule 12:

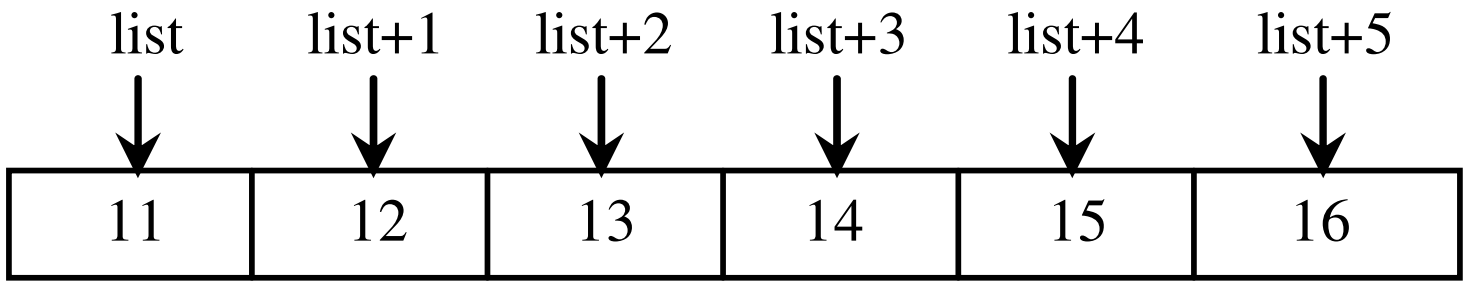
You can not change the content of the pointer pc NOR the content of the object it points to.

```
const int wrong=0;
pc=&wrong;        //WRONG
```

Arrays and Pointers

An array variable without a bracket and a subscript actually represents the starting address of the array. In this sense, an array variable is essentially a pointer. Suppose you declare an array of int value as follows:

int list[6] = {11, 12, 13, 14, 15, 16};



Array Pointer

*(list + 1) is different from *list + 1. The dereference operator (*) has precedence over +. So, *list + 1 adds 1 to the value of the first element in the array, while *(list + 1) dereference the element at address (list + 1) in the array.

ArrayPointer

PointerWithIndex

Passing Pointer Arguments

A pointer argument can be passed by value or by reference. For example, you can define a function as follows:

void f(**int*** p1, **int*** &p2)
which is equivalently to

typedef int* intPointer;
void f(intPointer p1, intPointer& p2)

Here p1 is pass-by-value and p2 is pass-by-reference.

TestPointerArgument

array parameter or pointer parameter

```
void m(int list[], int size)
```

can be replaced by

```
void m(int* list, int size)
```

```
void m(char c_string[])
```

can be replaced by

```
void m(char* c_string)
```

const parameter

If an object value does not change, you should declare it const to prevent it from being modified accidentally.

ConstParameter

Returning a Pointer from Functions

You can use pointers as parameters in a function. Can you return a pointer from a function? The answer is yes.

ReverseArrayUsingPointer

Useful Array Functions

C++ provides several functions for manipulating arrays. You can use the min_element and max_element functions to return the pointer to the minimal and maximal element in an array, the sort function to sort an array, the random_shuffle function to randomly shuffle an array, and the find function to find an element in an array. All these functions use pointers in the arguments.

UsefulArrayFunctions

Why Do We Need Dynamic Memory Allocation?

See this example

WrongReverse

CorrectReverse

Dynamic Memory Allocation

```
int* result = new int[6]; // Allocate

delete [] result; // Deallocate


int* p = new int; // Allocate

delete p; // Deallocate
```

Creating Dynamic Objects

You can also create objects dynamically on the heap using the following syntax:

```
ClassName* pObject = new ClassName(); or ClassName *pObject = new ClassName; or
ClassName* pObject = new ClassName(arguments);
```

```
// Create an object using the no-arg constructor
string* p = new string(); // or string* p = new string;
// Create an object using the constructor with arguments
string* p = new string("abcdedfg");
```

Accessing Dynamic Objects

To access object members via a pointer, you must dereference the pointer and use the dot (.) operator to object's members. For example,

```
string* p = new string("abcdedfg");
cout << "The first three characters in the string are "
  << (*p).substr(0, 3) << endl;
cout << "The length of the string is " << (*p).length() <<
endl;
```

Accessing Dynamic Objects

C++ also provides a shorthand member selection operator for accessing object members from a pointer: arrow (->) operator, which is a dash (-) immediately followed by the greater than (>) symbol. For example,

```
cout << "The first three characters in the string are "
  << p->substr(0, 3) << endl;
cout << "The length of the string is " << p->length()
<< endl;
```

What is wrong here!?

```
void Leaky() {
  int *x = new int(5);  // heap allocated
  (*x)++;
  std::cout << *x << std::endl;
}
```

What can go wrong here!?

```
#include <iostream>
void Test()
{
    int* pointer = new int[5];
   //manipulate the memory block
   //what may happen here!
  delete[] pointer;
}
```

Smart Pointers in C++

- A smart pointer is an object that stores a pointer to a heap allocated object
- A smart pointer looks and behaves like a regular C++ pointer
- How? by overloading * , -> , [ ] , etc.
- A smart pointer can help you manage memory
- The smart pointer will delete the pointed-to object at the right time
- When? that is depends on what kind of smart pointer you use, so if you use a smart pointer correctly, you no longer have to remember when to delete new'd memory

Why Smart Pointers?

- Many issues are involved with pointers like ensuring the lifetime of objects referred to by pointers, dangling references, and memory leaks.

```
void Leaky() {
  int *x = new int(5);  // heap allocated
  (*x)++;
  std::cout << *x << std::endl;
}
```

```
#include <iostream>
void Test()
{
    int* pointer = new int[5];
   //manipulate the memory block
   //what may happen here!
  delete[] pointer;
}
```

C++'s auto_ptr

- The auto_ptr class is part of C++'s standard library
  - it's useful, simple, but limited and replaced by alternatives in C++11 - but still used
- An auto_ptr object takes ownership of a pointer
  - Automatic cleanup,

    when the auto_ptr object is delete'd or falls out of scope, its destructor is invoked, just like any C++ object
  - Automatic initialization,

    you don't need to initialize the auto_ptr to NULL, since the default constructor does that for you.

C++'s auto_ptr

```
void NotLeaky() {
  std::auto_ptr<int> x(new int(5));  // wrapped, heap-allocated
  (*x)++;
  std::cout << *x << std::endl;
}
```

auto_ptr does not allow us to initialize an object with an ordinary pointer by using the assignment syntax. So, we must initialize the **auto_ptr** directly by using **its value**.

```
std::auto_ptr<int> x(new int); // RIGHT
std::auto_ptr<int> x = new int; // WRONG
```

auto_ptr limitations

- An auto_ptr can't point to an array. When deleting a pointer to an array we must use **delete[]** to ensure that destructors are called for all objects in the array, but auto_ptr uses **delete**.

```
std::auto_ptr<Test> p(new Test[5]); //wrong
```

- It can't be used with the STL containers- elements in an STL container.

auto_ptr limitations

auto_ptr transfers the ownership when it is assigned to another auto_ptr. This is really an issue while passing the auto_ptr between the functions.

Auto_ptr_pass_to_function

C++11 smart pointers

- shared_ptr
- weak_ptr
- unique_ptr

C++11 smart pointers

- shared_ptr : shared ownership. Multiple shared pointers can refer to a single object and when the last shared pointer goes out of scope, memory is released automatically.

```
int main( ) {
   shared_ptr<int> shared_pointer( new int );
   return 0;
}
```

C++11 smart pointers

- shared_ptr : allocates memory internally, to hold the reference count, You can use the helper function/macro make_shared( ) for the creation process.

```
int main( ) {
   shared_ptr<int> shared_pointer = make_shared<int>(100);
   return 0;
}
```

C++11 smart pointers

- shared_ptr : problem when shared_ptrs from different groups share the same memory block.

This is all good!

```
int main( ) {
   shared_ptr<int> sp1( new int );
   shared_ptr<int> sp2 = sp1;
   shared_ptr<int> sp3;
   sp3 = sp2;
   return 0;
}
```

int* p = new int;

Do NOT create a shared pointer from this!

```
int main( ) {
   int* p = new int;
   shared_ptr<int> sp1( p);
   shared_ptr<int> sp2( p );
   return 0;
}
```

C++11 smart pointers

- weak_ptr : A weak pointer provides sharing semantics and not owning semantics.
- This means a weak pointer can share a resource held by a shared_ptr. There are no * or -> for weak_ptr since it doesn't own the resource.

```
int main( ) {
    shared_ptr<int> sp( new int );
   weak_ptr<int> wp( sp );
   weak_ptr<int> wp1 = wp;
   return 0;
}
```

C++11 smart pointers

- unique_ptr : follows the exclusive ownership semantics, i.e. at any point of time, the resource is owned by only one unique_ptr. When unique_ptr goes out of scope, the resource is released. If the resource is overwritten by some other resource, the previously owned resource is released. So it guarantees that the associated resource is released always.
- Can not be copied to another **unique_ptr**, passed by value to a function, or used in any Standard Template Library (STL) algorithm that requires copies to be made.

```
int main( ) {
    unique_ptr<int> up( new int );
   unique_ptr<int[ ]> uptr( new int[5] );
   return 0;
}
```

The this Pointer

Sometimes you need to reference a class's hidden data field in a function. For example, a data field name is often used as the parameter name in a set function for the data field. In this case, you need to reference the hidden data field name in the function in order to set a new value to it. A hidden data field can be accessed by using the this keyword, which is a special built-in pointer that references to the calling object.

CircleWithThisPointer.cpp

Destructors

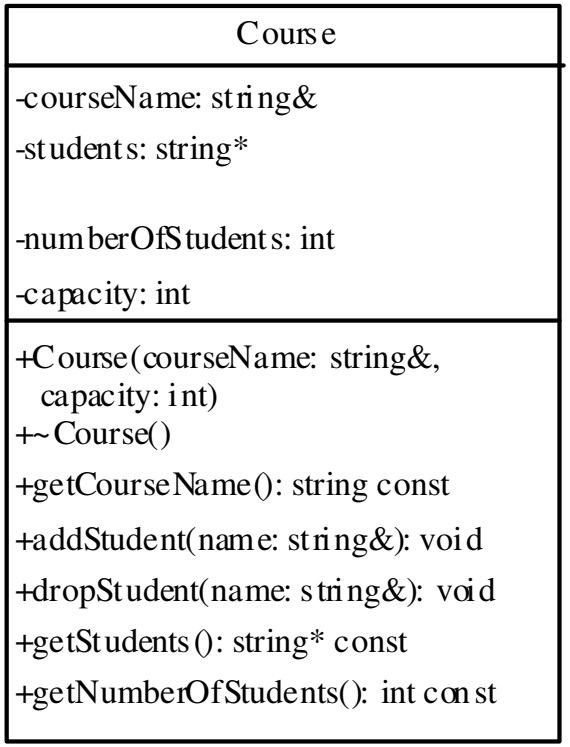
Destructors are the opposite of constructors. A constructor is invoked when an object is created and a destructor is invoked when the object is destroyed. Every class has a default destructor if the destructor is not explicitly defined. Sometimes, it is desirable to implement destructors to perform customized operations. Destructors are named the same as constructors, but you must put a tilde character (~) in front of it.

CircleWithDestructor.h

TestCircleWithDestructor.cpp

CircleWithDestructor.cpp

The Course Class



The name of the course.

An array of students who take the course. students is a pointer for the array.

The number of students (default: 0).

The maximum number of students allowed for the course.

Creates a Course with the specified name and maximum number of students allowed.

Destructor

Returns the course name.

Adds a new student to the course.

Drops a student from the course.

Returns the array of students for the course.

Returns the number of students for the course.

Course.h

Course.cpp

TestCourse

Copy Constructors

Each class may define several overloaded constructors and one destructor. Additionally, every class has a *copy constructor*. The signature of the copy constructor is:

ClassName(const ClassName&)

For example, the copy constructor for the Circle class is

Circle(const Circle&)

The copy constructor can be used to create an object initialized with another object's data. By default, the copy constructor simply copies each data field in one object to its counterpart in the other object.

CopyConstructorDemo

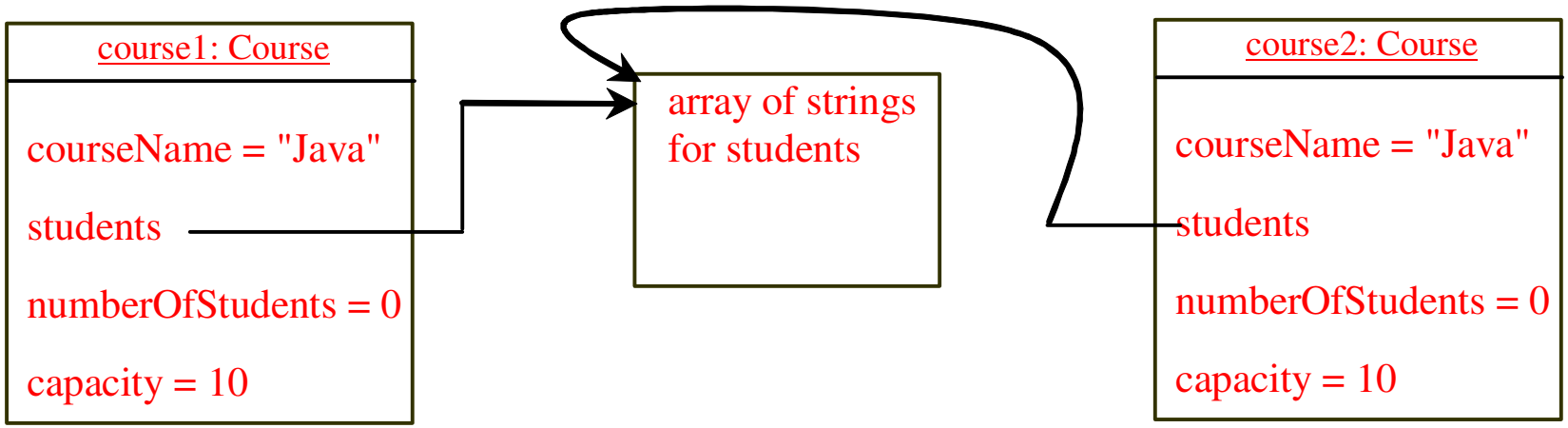
Shallow Copy vs. Deep Copy

The default copy constructor or assignment operator for copying objects performs a *shallow copy*, rather than a *deep copy*, meaning that if the field is a pointer to some object, the address of the pointer is copied rather than its contents.

ShallowCopyDemo

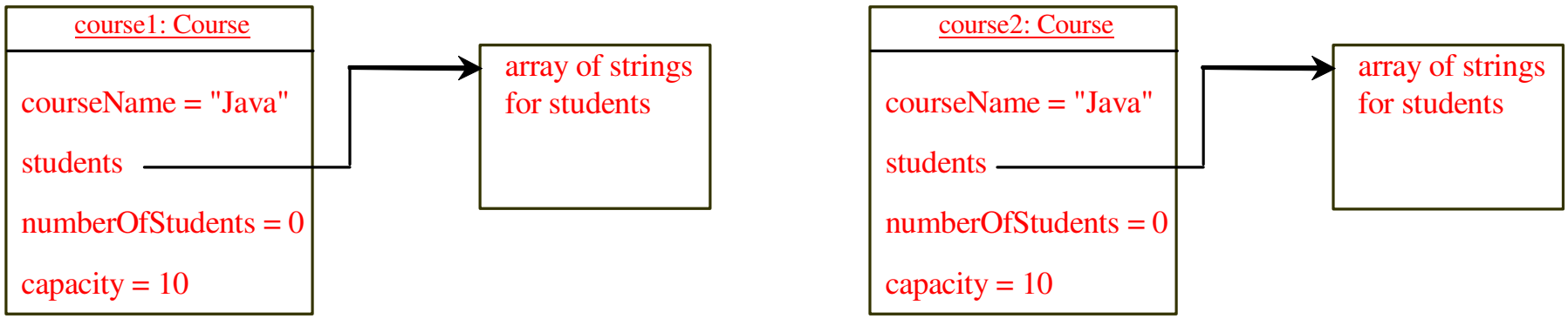
Shallow Copy

Copy course1 to course2. After copying course1 to course2, both course1 and course2 point to the same student.



Deep Copy

After course1 is copied to course2, the students data field of course1 and course2 point to two different arrays.



CourseWithCustomCopyConstructor.h

CoursewithCustomCopyConstructor.cpp

CustomCopyConstructorDemo