## Pointers and Memory Allocation

- The C++ run-time system can create new objects
- A *memory allocator* finds a storage location for a new object

```
new Employee;
```

- The memory allocator keeps a large storage area, called the *heap*
- The heap is a flexible pool of memory that can hold values of any type
- When you allocate a new heap object, the memory allocator tells where the object is located, by giving you the object's *memory address*
- Use a *pointer* to store and manipulate a memory address

## Deallocating Dynamic Memory

- The expression: `new Employee`
- is very different from: `Employee harry;`
- *harry* lives on a *stack*

- The *stack* is a storage area associated with the defining function

```
void f()

{

   Employee harry; // memory for employee allocated
on the stack

   ...
```

```
} // Memory for employee automatically reclaimed
```

- Values allocated from the heap stay alive until the programmer reclaims it

## Pointers and Memory Allocation

- The allocator returns an *address*, or *pointer*
- Pointers are stored in a pointer variable
- To declare pointers:

```
Employee* boss;

Time* deadline;
```

- The types `Employee*` and `Time*` are pointers to employee and time objects
- `boss` and `deadline` store addresses
- They do **not** store actual employee or time objects

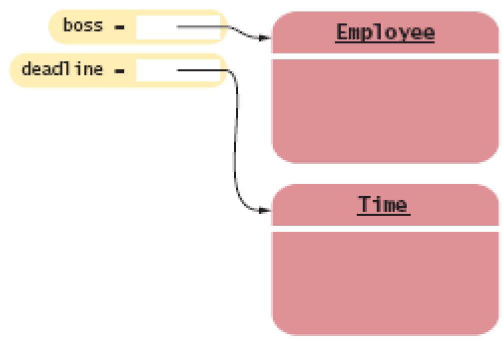



Figure 1 Pointers and the Objects to Which They Point

## Pointers and Memory Allocation

- You can also call the `new` command in conjunction with a constructor to initialize the object

```
Employee* boss = new Employee("Lin, Lisa", 68000);
```

- To access a value, given a pointer, you must *dereference* the pointer

```
Employee* boss = ...;
raise_salary(*boss, 10);
```

- To get the boss' name, you might try

```
string name = *boss.get_name(); // Error
```

- `.` has higher precedence; you tried to send the pointer itself a message
- This will get an `Employee` object, then get its name:

```
string name = (*boss).get_name(); // Error
```

- The `->` operator does the same thing:

```
string name = boss->get_name(); // Error
```

## Pointers and Memory Allocation

- The special value `NULL` indicates that a pointer doesn't point anywhere
- Never leave a pointer uninitialized
- Set them to `NULL` when you define them

```
Employee* boss = NULL; // will set later
. . .
if (boss != NULL) name = boss->get_name(); // OK
```

- You cannot dereference a `NULL` pointer

```
Employee* boss = NULL;

string name = boss->get_name(); // NO!! Program will
crash
```

- Crashing is better than processing erroneous data

```
Employee* boss;

string name = boss->get_name(); // NO!! boss
contains a random address
```

- Better still, test for the sentinel, as above

## Syntax : `new` Expression

```
new type_name
new type_name(expression1, expression2, ... ,
expressionn)
```

Example:

```
new Time;
new Employee("Lin, Lisa", 68000)
```

Purpose:

Allocate and construct a value on the heap and return a pointer to the value.

## Syntax : Pointer Variable Definition

```
type_name* variable_name;
type_name* variable_name = expression;
```

Example:

```
Employee* boss;
```

```
Product* p = new Product;
```

Purpose:

Define a new pointer variable, and optionally supply an initial value.

## Syntax : Pointer Dereferencing

```
*pointer_expression
pointer_expression->class_member
```

Example:

```
*boss
boss->set_salary(70000)
```

Purpose:

Access the object to which a pointer points.

## Common Error

# Declaring Two Pointers on the Same Line

- In this declaration, `p` is a pointer, while `q` is an actual `Employee`
- `Employee* p, q;`
- To make them both pointers:

- `Employee *p, *q;`(the spacing is irrelevant)
- Might be clearer to use a line for each declaration:
- `Employee *p;`
- `Employee *q;`

## Advanced

The `this` Pointer

- Every (non-static) method has a `this` pointer
- `this` is the pointer to the implicit parameter
- If you call

```
next.is_better_than(best)
```

  - `this` is of type `Product*`
  - `this` points to `next`
- Could be used like this:

```
bool Product::is_better_than(Product b)
{
   if (this->price == 0) return true;
   if (b.price == 0) return false;
```

```
   return this->score / this->price > b.score / 
b.price;
}
```

- Note, `b` is an object, `this` is a pointer

## Deallocating Dynamic Memory

- You must manually reclaim dynamically allocated objects
- Use the `delete` operator

```
void g()
{
   Employee* boss;
   boss = new Employee(...); // Memory 
for employee allocated on the heap
   ...
   delete boss; // Memory for employee 
manually reclaimed
}
```

- `delete` does nothing to `boss`
- `boss` is a stack variable — will be reclaimed at the end of the block
- `delete` frees the memory that `boss` pointed to
- `boss` is not set to `NULL`; it points to the same place

## Syntax : `delete` Expression

```
delete pointer_expression;
```

Example:

```
delete boss;
```

Purpose:

Deallocate a value that is stored on the heap and allow the memory to be reallocated.

## Common Error

Dangling Pointers

- A pointer that doesn't point to a valid object
  - Pointer wasn't initialized, or
  - Object pointer referenced was reclaimed
- Writing to this location may change other variables, or your program
- Reading from this location might crash your program (if you're lucky)
- This is particularly insidious:

```
delete boss;

string name = boss->get_name(); // NO!!
boss points to a deleted element
```

- Almost impossible to catch during testing
- Object appears to still be there
- Location might well be claimed for something else

## Common Error

Memory Leaks

- A memory block that is not deallocated is a *memory leak*
- Leaked memory can cause the heap to run out of memory
  - Program crashes
  - Computer freezes up
- Each `new` should be paired with a `delete`
- Memory leaks should be avoided, for memory-intensive or long-running programs
- Should be avoided for smaller programs, too

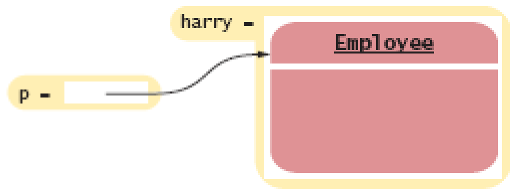
## Advanced Topic

The Address Operator

- The `&` operator (*address* operator) returns the address of an existing, stack variable

```
Employee harry;

Employee* p = &harry;
```




The Address Operator

- Never `delete` a stack variable!

```
delete &harry; // NEVER!
```

- That location would then be on the stack, *and* part of the heap memory

## Common Uses for Pointers

Optional Attributes

- Consider a department class, which allows for an optional receptionist:

```
class Department
{
    ...
private:
    string name;
    Employee* receptionist;
};
```

- `receptionist` points to an actual employee, or is `NULL` if not needed
- This is better than allocating space for an object that might not be used.

```
class Department // Modeled without
pointers
{
    ...
private:
    string name;
    bool has_receptionist;
    Employee receptionist;
};
```

## Common Uses for Pointers

Object Sharing

- Rather than duplicating objects, use pointers to share the object
- Example: In some departments, the secretary and the receptionist are the same person

```
class Department
{
    ...
private:
    string name;
```

```
    Employee* receptionist;
    Employee* secretary;
};
```

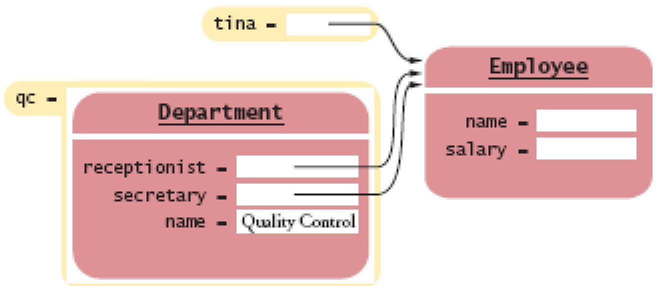



Figure 3 Three Pointers Share an Employee Object

```
...
Employee* tina = new Employee("Tester,
Tina", 50000);
Department qc("Quality Control");
qc.set_receptionist(tina);
qc.set_secretary(tina);
tina->set_salary(55000);
```

## Common Uses for Pointers

Sharing Objects (cont.)

- Particularly important when changes to the object need to be observed by all users of the object
- Without using pointers, changing Tina's salary would not update the information in the receptionist or secretary attribute

```
Employee tina("Tester, Tina", 50000);
Department qc("Quality Control");
qc.set_receptionist(tina);
qc.set_secretary(tina);
tina.set_salary(55000);
```

- Department object now contains two copies of Tina
- Copies are not affected by Tina's raise

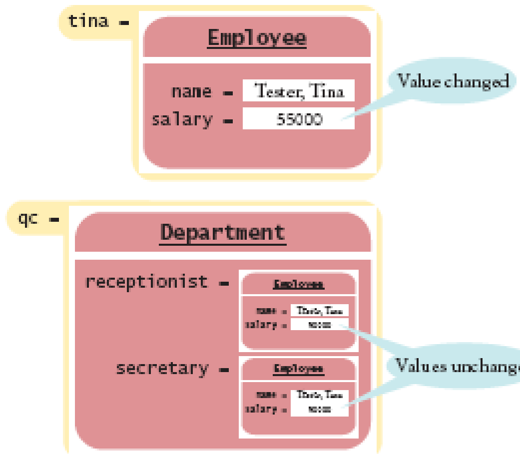



Figure 4 Separate Employee Objects

```
#include <string>
#include <iostream>

using namespace std;

#include "ccc_empl.h"

/**
   A department in an organization.
*/
class Department
{
public:
   Department(string n);
   void set_receptionist(Employee* e);
   void set_secretary(Employee* e);
   void print() const;
private:
   string name;
   Employee* receptionist;
   Employee* secretary;
```

```
};

/**
   Constructs a department with a given name.
   @param n the department name
*/
Department::Department(string n)
{
   name = n;
   receptionist = NULL;
   secretary = NULL;
}

/**
   Sets the receptionist for this department.
   @param e the receptionist
*/
void Department::set_receptionist(Employee* e)
{
   receptionist = e;
}

/**
   Sets the secretary for this department.
   @param e the secretary
*/
void Department::set_secretary(Employee* e)
{
   secretary = e;
}

/**
   Prints a description of this department.
*/
void Department::print() const
{
   cout << "Name: " << name << "\n"
        << "Receptionist: ";
   if (receptionist == NULL)
      cout << "None";
   else
      cout << receptionist->get_name() << " "
           << receptionist->get_salary();
   cout << "\nSecretary: ";
   if (secretary == NULL)
      cout << "None";
   else if (secretary == receptionist)
      cout << "Same";
   else
      cout << secretary->get_name() << " "
           << secretary->get_salary();
   cout << "\n";
}

int main()
{
   Department shipping("Shipping");
```

```
    Department qc("Quality Control");
    Employee* harry = new Employee("Hacker, Harry", 45000);
    shipping.set_secretary(harry);
    Employee* tina = new Employee("Tester, Tina", 50000);
    qc.set_receptionist(tina);
    qc.set_secretary(tina);
    tina->set_salary(55000);
    shipping.print();
    qc.print();
    delete tina;
    delete harry;
    return 0;
}
```

## Advanced Topic

References

- You saw reference parameters.

```
void raise_salary(Employee& e, double by)
{
    double new_salary = e.get_salary() * (1 + by / 100);
    e.set_salary(new_salary);
}
```

- The value of `harry` may change in this call:

```
raise_salary(harry, percent);
```

- References are just syntactic sugar for pointers
- This function receives the address of an `Employee` object, and a copy of a `double`

## Advanced Topic (cont.)

References

- In C this function would've been written:

```
void raise_salary(Employee* pe, double
by)
{
   double new_salary = pe->get_salary() *
(1 + by / 100);
   pe->set_salary(new_salary);
}
```

- The call, above, would look like this:

```
raise_salary(&harry, percent);
```

- When you use references, the compiler takes care of referencing and dereferencing pointers.

## Arrays and Pointers

- There is an intimate connection between arrays and pointers in C++
- The name of an array is a pointer to the starting element

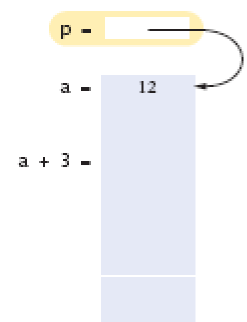
```
int a[10];

int* p = a; // now p points to a[0];
```

- `a` can be dereferenced: `*a = 12;` is the same as `a[0] = 12;`
- Pointers into arrays support *pointer arithmetic*: `*(a + 3)` is the same as `a[3]`

## Arrays and Pointers

- This relationship is called the *array/pointer duality law*
- For any integer `n`,
- `*(a + n) ≡ a[n]`
- This explains why array indices start at 0
- `a` (`a+0`) points to the start of the array




Pointers into an Array

## Arrays and Pointers

- When an array is passed into a function, it is actually a pointer to the starting element of the array

```
double maximum(const double a[], int
a_size)
{
   if (a_size == 0) return 0;
   double highest = a[0];
```

```
    for (int i = 0; i < a_size; i++)
        if (a[i] > highest)
            highest = a[i];
    return highest;
}
```

- The function receives only the starting address of the array

```
double maximum(const double* a, int
a_size)
{
    // Identical code as above yields same
results
    ...
}
```

**Advanced Topic**

## Using Pointers to Step Through an Array

- Rather than incrementing an index, increment the pointer

```
double maximum(const double* a, int
a_size)
{
    if (a_size == 0) return 0;
    double highest = *a;
    const double* p = a + 1;
    int count = a_size - 1;
    while (count > 0)
```

```
    {
       if (*p > highest)
          highest = *p;
       p++;
       count--;
    }
    return highest;
 }
```

## Common Error

Returning a Pointer to a Local Array

- Don't return pointers to local (stack) variables

```
double* minmax(const double a[], int
a_size)
{
   assert(a_size > 0);
   double result[2];
   result[0] = a[0]; // result[0] is the
minimum
   result[1] = a[0]; // result[1] is the
maximum

   for (int i = 0; i < a_size; i++)
   {
      if (a[i] < result[0]) result[0] =
a[i];
```

```
        if (a[i] > result[1]) result[1] =
a[i];
    }
    return result; // ERROR!
}
```

- `result` is local to `minmax`
- When function exits, `result` is gone

## Advanced Topic

Dynamically Allocated Arrays

- You can allocate arrays from the heap:

```
int staff_capacity = ...;
Employee* staff = new
Employee[staff_capacity];
```

- `new[]` operator allocates an array of `staff_capacity` `Employee`s (using default constructor)
- Size does **not** need to be known at compile time
- Manipulated just like any other array
- This is how variable-sized containers, like the `Vector`, is implemented
- **Must be deallocated (reclaimed)** using the `delete[]` operator:

```
delete[] staff;
```

## Advanced Topic (cont.)

### Dynamically Allocated Arrays - Resizing

- If later you need a larger array:
  - get larger array from the heap
  - copy the contents over
  - delete the original array
  - fix up your pointers:

```
int bigger_capacity = 2 * staff_capacity;

Employee* bigger = new Employee[bigger_capacity];

for (int i = 0; i < staff_capacity; i++)

   bigger[i] = staff[i];

delete[] staff;

staff = bigger;

  staff_capacity = bigger_capacity;
```

## Pointers to Character Strings

- C++ inherits primitive string handling from the C language, in which strings are represented as arrays of `char` values
- Though not recommended for use, you'll need to recognize character pointers or arrays in your programs when you see them
- Literal strings are stored inside `char` arrays

```
char s[] = "Harry";
```

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 'H' | 'a' | 'r' | 'r' | 'y' | '\0' |

- Space for the null-terminator (`\n`) is automatically allocated

## Pointers to Character Strings

- Many pre-STL functions return a `char*`
- Use constructor `string(char *)` to convert any character pointer or array to a safe and convenient `string` object:

```
char* p = "Harry";

string name(p);
```

- Some functions require a `char*` as an argument
- The `string::c_str` method returns a `char*` that points to the first character in the string object
- E.g., `tempnam()`, in the standard library, yields the name of a temporary file, and expects a `char*` parameter for the directory name:

```
string dir = ...;
char* p = tempnam(dir.c_str(), NULL);
```

## Common Error

Failing to Allocate Memory

- Writing (or copying) a string to random memory is a *very* common and dangerous error

```
char* p;
strcpy(p, "Harry");
```

- This is **not** a syntax error

- If you're lucky, the address is not legal, and the program crashes
- If you're less lucky, the data will be written wherever
- This is a very insidious error; tough to detect, and tough to find
  - It might be corrupting somebody else's memory
  - Somebody else might be overwriting "your" string

## Common Error

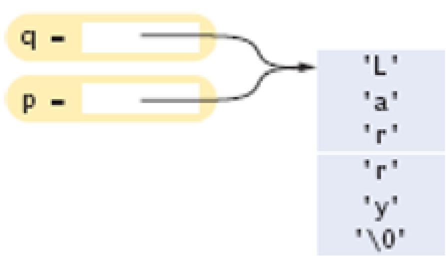
Copying Character Pointers

- Assignment, copying and comparing `string` objects is intuitive:

```
string s = "Harry";
string t = s;
t[0] = 'L'; // now s is "Harry" and t is "Larry"
```

- `s` and `t` are distinct objects

- Same example, using pointers:




```
char* p = "Harry";
```

```
char* q = p;
q[0] = 'L'; // Now
both p and q point to
"Larry"
```

Two Character Pointers into the Same Character Array

- `p` and `q` are distinct pointers, storing the same address
- Both refer to the same object

## Common Error (cont.)

Copying Character Pointers

- Arrays can **not** be assigned in the usual way:

```
char a[] = "Harry";
char b[6];
b = a; // ERROR
```

- Use `strcpy()`:
- `strcpy(b, a);`
- Since `strcpy()` has no idea how large array `b` might be, this is safer:

```
strcpy(b, a, 5);
```

## Pointers to Functions

- Sometimes a function depends on another function
  - Consider a function that prints a table of values of the function $f(n) = n^2$ :

| | |
|---|---|
| 1 | 1 |
| 2 | 4 |
| 3 | 9 |
| 4 | 16 |
| … | |
| 10 | 100 |

  - Same logic to print the values of $f(x) = x - 2$
  - Function `print_table` takes a *function pointer* as an argument
  - As with arrays, the name of a function is really a pointer to a function:

```
print_table(sqrt);
```

## Pointers to Functions

- To print a table of squares, first make a `square` function:

```
double square(double x) { return x * x; }
...
print_table(square);
```

- The function to print a table:

```
void print_table(DoubleFunPointer f)
{
   cout << setprecision(2);
   for (double x = 1; x <= 10; x++)
   {
      double y = f(x);
      cout << setw(10) << x << "|" <<
setw(10) << y << endl;
   }
}
```

  - `DoubleFunPointer` will be explained shortly

## Pointers to Functions

- The parameter `f` can be used as any other function
- Some prefer to call the function like this:

```
(*f)(x)
```

- To declare the function pointer:

```
double (*f)(double)
```

- This is a function (not a pointer) which returns a `double*` :

```
double *f(double)
```

- `print_table()` looks like this:

```
void print_table(double (*f)(double))
```

- A *type definition* makes this easier to read:

```
typedef double
(*DoubleFunPointer)(double);
void print_table(DoubleFunPointer f);
```