

C++ Structures & Pointers

Wagner Truppel
Lecturer, Dept. of Computer Science & Engineering
UC Riverside

wagner@cs.ucr.edu
<http://www.cs.ucr.edu/~wagner>

<http://www.cs.ucr.edu/cs12>

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 1

Today's Topics

- C++ Structures
- Pointers
- Static and Dynamic Memory Allocation
- Dynamic arrays

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 2

C++ structures

- Section 6.1 of Savitch's book
- We've seen
 - ◆ Built-in data types
 - * **int, float, double, bool, char**, etc
 - * They are very simple
 - ◆ Arrays
 - * Also "built-in" (kind of...)
 - * Collection of same-type elements
- Is it possible to have a collection of **different**-type elements that we can treat as a single entity ?
- Yes! But why would you need or want a creature like that ?

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 3

C++ structures

- Is it possible to have a collection of *different*-type elements that we can treat as a single entity ?
- Yes! But why would you need or want a creature like that ?
 - ◆ Easier explained with an example
- Suppose you have to write an employee management program
- You need to have some data structure to store employees and their data
- Each employee has
 - ◆ First and last names
 - ◆ Social Security Numbers
 - ◆ Address and phone numbers
 - ◆ Salary, etc

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 4

C++ structures

- Each employee has
 - ◆ First and last names
 - ◆ Social Security Numbers
 - ◆ Address and phone numbers
 - ◆ Salary, etc
- These are data of different types
 - ◆ Names are strings
 - ◆ SSNs are integer numbers
 - ◆ Addresses are strings
 - ◆ Phone numbers are integer numbers
 - ◆ Salary are real numbers
- It would be nice to be able to refer to an employee and his/her information directly, as a single entity

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 5

C++ structures

- It would be nice to refer to an employee and his/her information directly, as a single entity
- Example: instead of having
 - ◆ An array of first names
 - ◆ An array of last names
 - ◆ An array of SSNs
 - ◆ An array of addresses
 - ◆ An array of phone numbers
 - ◆ An array of salaries
- It's far more convenient to have simply
 - ◆ An array of Employees
- C++ lets you do that with something called a structure

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 6

C++ structures

```

struct Employee
{
    std::string firstName;
    std::string lastName;
    int ssNumber;
    Address address;
    int phoneNumber;
    double salary;
}; // ⚡ do not forget this ';'

struct Address
{
    std::string street;
    int number;
    std::string city;
    std::string state;
    int zip;
}; // ⚡ do not forget this ';'

```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 7

C++ structures

- Note: you can have a struct inside of another struct
- How do you access a struct's **member variable** ?

```

Employee wagner;
wagner.firstName = "Wagner";
wagner.lastName = "Truppel";
wagner.ssNumber = 123456789; // not really...
wagner.phoneNumber = 9095551234; // not really...
wagner.address.zip = 92507;
wagner.address.state = "CA";

```

- Why is the semi-colon ";" required after the {} ?
 - ◆ Because you can declare structure variables at the same time you define your structures

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 8

C++ structures

- Why is the semi-colon ";" required after the {} ?
 - ◆ Because you can declare structure variables at the same time you define your structures

```

struct Employee
{
    std::string firstName;
    std::string lastName;
    int ssNumber;
    Address address;
    int phoneNumber;
    double salary;
} wagner, brian, peter;

```

- The above declares 3 Employee variables but their member variables haven't been assigned yet.

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 9

C++ structures

- Functions can take structures as arguments !
 - ◆ using call by value:


```
int getSSNumber(Employee emp)
{ return emp.ssNumber; }
```
 - ◆ using call by reference:


```
void setSalary(Employee &emp, double salary)
{ emp.salary = salary; }
```
 - ◆ What's the difference once again ?
- Functions can return structures too !


```
Address getAddress(Employee &emp)
{ return emp.address; }
```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 10

C++ structures

- Initializing structures
 - ◆ Similar to how you can initialize static arrays

```
struct Date
{
    int month; // 1 = Jan, ..., 12 = Dec
    int day; // 1..31
    int year; // 2000, 2001, etc
};

Date myBirthday = { 3, 25, 1703 };
// yes, I am 300 years old !! :-)
```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 11

C++ structures

- Useful example: a **stack** of int's

```
struct Stack
{
    int top; // -1 indicates empty stack
    int array[MAX_STACK_SIZE];
};

void push(Stack &s, const int value)
{ s.array[++s.top] = value; }

int pop(Stack &s)
{ return s.array[s.top--]; }
```

- These functions apply to **any** stack now !

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 12

Pointers

- Secs. 10.1 & 10.2 of Savitch's book
- A little review of how variables are stored in memory

...									
353									
352									
351									
350	0	1	0	1	1	0	1	0	
349	0	0	0	0	0	0	0	0	
348	0	0	0	0	0	0	0	0	
347	0	0	0	0	0	0	0	0	
346	0	1	1	1	1	0	1	1	
345									
...									

char c = 'Z'; \x90 = \x5A

int x = 123;

address(c) = 350
address(x) = 346

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 13

Pointers

- Can we refer to a variable's **address** rather than to its **value** ?
- Yes ! The variable's address is stored in another variable, a **pointer variable**.

...									
353									
352									
351									
350	0	1	0	1	1	0	1	0	
349	0	0	0	0	0	0	0	0	
348	0	0	0	0	0	0	0	0	
347	0	0	0	0	0	0	0	0	
346	0	1	1	1	1	0	1	1	
345									
...									

char c = 'Z'; \x90 = \x5A

int x = 123;

address(c) = 350
address(x) = 346

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 14

Pointers

- Pointer variables are **type-dependent**

```

int *px;
px = &x; \x px = 346 (sort of...)

char *pc;
pc = &c; \x pc = 350 (sort of...)
    
```

...									
353									
352									
351									
350	0	1	0	1	1	0	1	0	
349	0	0	0	0	0	0	0	0	
348	0	0	0	0	0	0	0	0	
347	0	0	0	0	0	0	0	0	
346	0	1	1	1	1	0	1	1	
345									
...									

char c = 'Z'; \x90 = \x5A

int x = 123;

address(c) = 350
address(x) = 346

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 15

Pointers

- A pointer variable holds the **address** of the variable it points to (plus type information)
- Details are platform dependent
- What matters is that a pointer variable points to a (regular) variable

```
int x = 123; // declares an int variable x and sets its value to 123
int *px; // declares a pointer variable px pointing to an int
px = &x; // sets the pointer's target to the int variable x
```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 16

Pointers

- The notation is confusing at first...


```
int x; // declares a regular variable
int *px; // declares a pointer variable
px = &x; // sets the target of the pointer
*px = 456; // sets the value of the variable pointed to by px (in this case, x)
```
- So...
 - type* ***pointer declares** a pointer variable pointing to a variable of type *type*
 - &var** returns the **address** of variable *var* (**addressing operator**)
 - *pointer accesses the value** of the variable pointed to by *pointer* (**dereferencing operator**)

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 17

Pointers

- A surprising – but typical – example: what's the output of this block of code ?


```
char c = 'A';
std::cout << c << std::endl;
char *p;
p = &c;
std::cout << *p << std::endl;
*p = 'R';
std::cout << *p << std::endl;
std::cout << c << std::endl;
```
- The output is:
 - A
 - A
 - R
 - R

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 18

Pointers

- You're used to declaring and setting a regular variable in one step:
`int x = 123;`
- You've just learned to declare and then define a pointer variable, in two steps:
`int *p;`
`p = &x;`
- Can you declare and define a pointer variable in one single step too ?
Yes, but it's a bit counter-intuitive...
`int *p = x; // is it like this ?`

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 19

Pointers

- You're used to declaring and setting a regular variable in one step:
`int x = 123;`
- You've just learned to declare and then define a pointer variable, in two steps:
`int *p;`
`p = &x;`
- Can you declare and define a pointer variable in one single step too ?
Yes, but it's a bit counter-intuitive...
`int *p = x; // is it like this ? No !`
`int *p = &x; // it's like this !`

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 20

Pointers

- More tricky stuff...
- What happens when you write
`p1 = p2; ?`

The diagram illustrates the state of two pointers, p1 and p2, and the memory locations they point to. On the left, labeled 'before', p1 points to a memory location containing 'A' (c1), and p2 points to a memory location containing 'R' (c2). On the right, labeled 'after', p1 now points to the memory location containing 'R' (c2), and p2 still points to the memory location containing 'R' (c2). This shows that p1 = p2; results in p1 pointing to the same memory as p2.

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 21

Pointers

- More tricky stuff...
- What about when you write `*p1 = *p2;` ?

before
`*p1 = *p2`

p1 → ['A']
c1

p2 → ['R']
c2

after
`*p1 = *p2`

p1 → ['R']
c1

p2 → ['R']
c2

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 22

Pointers

before
`p1 = p2`

p1 → ['A']
c1

p2 → ['R']
c2

after
`p1 = p2`

p1 → ['A']
c1

p2 → ['R']
c2

before
`*p1 = *p2`

p1 → ['A']
c1

p2 → ['R']
c2

after
`*p1 = *p2`

p1 → ['R']
c1

p2 → ['R']
c2

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 23

The new operator

- What are pointers useful for anyway ?
- Recall that all variables you're used to defining so far (other than globals) are allocated in the **stack area**
 - ◆ they're always local variables in some block of code or function
- How do you allocate variables in the **heap area** ?
- You use the operator **new**
- **new always returns a pointer**, so that's the answer: pointer variables are useful b/c they're the means by which you refer to variables allocated in the heap

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 24

The *new* operator

- `int *p; // declares a pointer to an int`
- `p = new int;`
- The above line of code:
 - ◆ Allocates space in the heap for an int
 - ◆ Makes `p` point to that int
 - ◆ But does **not** set that int's value (`*p` is **undefined**)
- `p = new int(123); // same but also sets the value of the new int`
`*p equals 123`

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 25

The *new* operator

- You can use **new** to allocate *any* kind of variable

```

struct Employee
{
    std::string name;
    int ssn;
};

Employee *empPtr =
    new Employee("John Doe", 123456789);
    
```

- How to access member variables ?

```

int hisSSN = (*empPtr).ssn; // these two ways
int hisSSN = empPtr->ssn; // are equivalent
    
```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 26

Static and Dynamic Memory Allocation

- "Regular" variables are typically allocated in the **stack** area – **always** local to some block of code
- The variables created with **new** are **always** allocated in the **heap** area
- Variables allocated in the stack have a well defined life span – they get popped off the stack when the block of code ends
- Variables allocated in the heap exist until someone deletes them – it's **your** responsibility to delete them !
- What if you don't ? **Memory leaks !**

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 27

The *delete* operator

- You use **delete** to de-allocate a variable created with **new**

```
Employee *empPtr =
    new Employee("John Doe", 123456789);

delete empPtr; // now *empPtr is undefined
```

- The space in the heap previously reserved for the employee record above is now available for other variables
- That means: **any** other pointers to that same area are **also** undefined
- Undefined pointers are also called **dangling pointers**

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 28

Dangling pointers

- If you try this:

```
int *p1 = new int(5);
int *p2 = p1; // p2 points to the same int
std::cout << *p1 << std::endl; // prints 5
std::cout << *p2 << std::endl; // also prints 5

delete p1; // that int is now gone !

std::cout << *p1 << std::endl;
```

- The result is **unpredictable**
- This will also be unpredictable:


```
std::cout << *p2 << std::endl;
```
- C++ has **no** built-in method to determine if a pointer is dangling

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 29

More on *new* & *delete*

- “**Regular**” variables are allocated in the **stack** area
- **Dynamic** variables are **allocated** in the **heap** using **new**, and **deleted** from the heap using **delete**
- What happens when you call new and there’s no more space in the heap ?
- New fails...
 - ◆ Older compilers:
 - A failing new returns **NULL**
 - ◆ Newer compilers following the standard definition of C++ :
 - a failing new raises an **exception** (which, if not dealt with, terminates the program)
 - we’ll talk about exceptions later in the course
- **NULL** is a special value... **any** pointer can be set to **NULL**, regardless of the pointer type

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 30

NULL & dangling pointers

- A good practice to decrease the chances of running into dangling pointers is to
 - ◆ set **all** pointers to dynamic variables you're done with to **NULL**
 - ◆ then you can test for **NULL** pointers

```

            if (p == NULL)
            { /* p is dangling */ }
            else
            { /* p is ok */ }
            
```
- Java is nicer than C++ when it comes to memory management
 - ◆ It automatically deletes dynamic variables when no one points to them
 - ◆ There's a price though: speed
 - ◆ But you still need to test for dangling pointers

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 31

More syntax: typedef

- It's often useful to create aliases ("nicknames") for types that you create
- This is particularly useful for pointer variables


```

            typedef int* IntPtr;
            
```
- `int*` and `IntPtr` are identical type names and can be used in each other's place


```

            IntPtr p = new int(123); // completely
            int *p = new int(123); // equivalent
            
```
- This has two major advantages:
 - ◆ You declare your pointer variables just as you declare regular variables (no * to worry about)
 - ◆ The other has to do with function arguments (more on this later)

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 32

Pointers & functions

- Functions can have pointer arguments
- Functions can return pointers too !


```

            char* SomeFunction(char* p);
            
```
- Note that even though `p` is a pointer, it's passed by value
- This passes the pointer `p` by reference


```

            char* SomeFunction(char*& p);
            
```
- Confusing... used typedef instead

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 33

Pointers & functions

- This passes the pointer p by reference
`char* SomeFunction(char*& p);`
- Confusing... use typedef instead
`typedef char* CharPtr;`
`char* SomeFunction(CharPtr& p);`
`CharPtr SomeFunction(CharPtr& p);`

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 34

Pointers & functions

- Alertness check: what's the output of the code section below ?

```

typedef char* CharPtr;
void f(CharPtr pc) { *pc = 'Z'; }
CharPtr p = new char('A');
f(p);
Std::cout << *p << std::endl;
    
```

- Note that pc is a call-by-value argument

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 35

Pointers & functions

```

typedef char* CharPtr;
void f(CharPtr pc) { *pc = 'Z'; }
CharPtr p = new char('A');
f(p);
Std::cout << *p << std::endl;
    
```

- Note that pc is a call-by-value argument, so p itself gets copied into the local variable pc inside the function
- Being a copy of p, that local variable **still** points to the same variable as p
- So the result is that the value of the variable pointed to by p changes from 'A' to 'Z' despite the fact that p is not passed by reference
- p never changed, but the value of its target did !

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 36

Pointers & functions

before function call: p points to 'A'

on entry to the function: p points to 'A', pc points to 'A'

inside the function: p points to 'Z', pc points to 'Z'

after the function call: p points to 'Z'

```
typedef char* CharPtr;
void f(CharPtr pc) { *pc = 'Z'; }
CharPtr p = new char('A');
f(p);
Std::cout << *p << std::endl;
```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 37

Pointers & Static Arrays

- Recall that array variables are interpreted as addresses to the first array element
- So array variables are pointer variables !
- Well, not quite, but almost.
- For all practical purposes in CS 12, you can think of array variables as pointer variables

```
int intA[100]; // static array !
int *p; // declares pointer to an int

p = intA; // this is ok !!!
p[0] = 7;
```

- That's the same as `intA[0] = 7;`

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 38

Dynamic Arrays

- You define a dynamic array the same way you define a pointer for a dynamic variable


```
char *vowels = new char[5];
```
- Compare with the static version


```
char vowels[5];
```
- Recall that I said once that functions cannot return arrays but that there's a way around it... Here it is:


```
char* f();
```
- This function returns a pointer to a **char** and that's what an array of **chars** is !

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 39

Dynamic Arrays

- Since dynamic arrays are dynamic variables, how do you delete them from the heap ?
- Using the **delete** operator...
- But the syntax is a little different than what we saw before:
 - ◆ **delete p;** // deletes the variable pointed to by the pointer p
 - ◆ **delete [] p;** // deletes the array variable p

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 40

Pointer Arithmetic

- Since a pointer is type dependent, it knows how large its underlying type is
- An array variable is a pointer variable so we can perform some arithmetic on the pointer variable itself
- `float *speed = new float[10];`
- `speed` is the address of `speed[0]`
- `speed + 1` = address of `speed[1]`
- ...
- `speed + 6` = address of `speed[6]`
- `speed` is an address
- addresses are numbers
- But `(speed + n)` does not follow the usual arithmetic: what gets added to the address in `speed` is `n` times the size of the base type

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 6 41
