

Compound types, compound data

Michael Nowak
Texas A&M University

Notes

Overview

Review

Compound types
References
Pointers
Arrays

Compound data
Vectors

References

Notes

Overview

Review

Compound types
References
Pointers
Arrays

Compound data
Vectors

References

Notes

Basic terminology

- Type** Defines a set of possible values and a set of operations for an object
- Object** Memory that holds a value of a given type
- Value** Set of bits in memory interpreted according to type
- Variable** Named object
- Declaration** Statement that gives a name to an object
- Definition** Declaration that sets aside memory for an object

Notes

Types

- ▶ C++ provides a set of types
 - ▶ `bool`, `char`, `int`, `double`, etc.
 - ▶ Called **built-in types**
- ▶ C++ programmers can define new types
 - ▶ Called **user-defined types**
 - ▶ We'll get to these eventually
- ▶ The C++ standard library provides a set of types; for example,
 - ▶ `std::string` is a variable-length sequence of characters
 - ▶ `std::vector` is a collection of objects, all of which have the same type; often referred to as a **container** because it "contains" other objects

Notes

Types

- ▶ The **type** of a **variable** determines which operations are valid and what their meanings are for that **type**
- ▶ **Integers and floating-point numbers**
 - ▶ `cin >>` reads a number
 - ▶ `cout <<` writes
 - ▶ `+` adds
 - ▶ `+= n` increments the value by `n`
 - ▶ `++` increments by 1
 - ▶ `-` subtracts
 - ▶ ...
- ▶ **Strings**
 - ▶ `cin >>` reads a word
 - ▶ `cout <<` writes
 - ▶ `+` concatenates
 - ▶ `+= s` adds the string `s` at the end
 - ▶ `++` is a type error
 - ▶ `-` is a type error
 - ▶ ...

Notes

Declarations

- ▶ A declaration is comprised of four parts:
 - ▶ An optional specifier
 - ▶ An initial keyword that specifies some non-type attribute
 - ▶ E.x., `const`, `extern`, etc.
 - ▶ A base type
 - ▶ A declarator
 - ▶ Composed of a name and optionally some declarator operators that are either prefix or postfix; most common declarator operators include:

*	pointer	prefix
*const	constant pointer	prefix
&	reference	prefix
[]	array	postfix
()	function	postfix
 - ▶ Postfix declarator operators bind more tightly than prefix ones
 - ▶ An optional initializer

Notes

Overview

Review

Compound types

References

Pointers

Arrays

Compound data

Vectors

References

Notes

Compound types

- ▶ A **compound type** is a type that is defined in terms of another type
- ▶ C++ has several, two of which we'll cover today:
 - ▶ References
 - ▶ Pointers
- ▶ The declarators that we have seen so far have been composed of only names, with the type of such variables the **base type** of the declaration
- ▶ More complicated declarators specify variables with compound types that are built from the base type of the declaration

Notes

Overview

Review

Compound types

References

Pointers

Arrays

Compound data

Vectors

References

Notes

References

- ▶ A reference creates an alias for an object, allowing indirect access to that object
- ▶ We can bind a reference to an object by writing a declarator of the form `&r`, where `r` is the name being introduced; for example,

```
int i = 11
```

```
int &r = i;
```

- ▶ The names `i` and `r` refer to the same object
- ▶ A reference is not an object but another name for an already existing object

Notes

References

- ▶ When declaring a variable of a primitive built-in type, the value of the `initializer` is copied into the object created
- ▶ When defining a reference to an object, we bind that reference to its `initializer`
- ▶ A reference cannot be rebound to some other object; because of this, all references must be `initialized`
- ▶ We cannot bind a reference to a literal:

```
int &a = 11; // error
```

however, we are allowed to take a `const &` to it:

```
const int &i = 11; // okay
```

Notes

Overview

Review

Compound types

References

Pointers

Arrays

Compound data

Vectors

References

Notes

Pointers

- ▶ A pointer is a compound type that "points to" another type
- ▶ A pointer's value is memory address of the object to which it points
 - ▶ You can think of these memory addresses as a an integer value
- ▶ We can define a pointer by writing a declarator of the form `*p`, where `p` is the name being defined, for example:

```
int i = 7;  
  
int *p = &i;
```

- ▶ We got the address of `i` by using the address-of operator (`&`)

Notes

Pointers

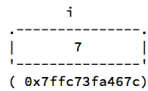
- ▶ A pointer points to an object of a given type
 - ▶ E.g., a `int *` points to an object whose type `int`
- ▶ A Pointer's type determines how the memory referred to by the pointer's value is used
 - ▶ E.g., what a `double *` points to can be added, but not concatenated, etc.
- ▶ The types of the pointer and object to which it points must match (there are *two exceptions*; we will get to them later)

Notes

Pointers

- When we declare a new variable, the identifier refers to an object that is created in memory with the value specified by the initializer:

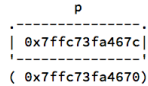
```
int i = 7;
```



A diagram showing a memory box labeled 'i' containing the value '7'. Below the box, the memory address is given as (0x7ffc73fa467c).

- We declare a pointer to `i` as follows:

```
int *p = &i;
```

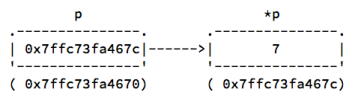


A diagram showing a memory box labeled 'p' containing the address '0x7ffc73fa467c'. Below the box, the memory address is given as (0x7ffc73fa4670).

Notes

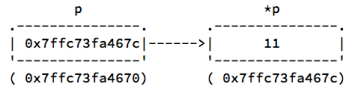
Pointers

- We can visualize this relationship informally as,



- To assign the value 11 to the object identified by `i` indirectly through `p`, we would write:

```
*p = 11;
```



A diagram showing two memory boxes. The first box is labeled 'p' and contains the address '0x7ffc73fa467c'. The second box is labeled '*p' and contains the value '11'. A dashed arrow points from the box 'p' to the box '*p'. Below each box, the memory address is given: (0x7ffc73fa4670) for 'p' and (0x7ffc73fa467c) for '*p'.

- We get the object to which the pointer points by using the dereference operator (`*`)

Notes

Important to note:

- `&` and `*` are used both as an operator in expressions and as part of a declaration to form compound types
- Make sure that you understand that it is *the context in which these symbols are used that determines their meaning*

Notes

Overview

Review

Compound types

References

Pointers

Arrays

Compound data

Vectors

References

Notes

Arrays

- ▶ An array is a homogeneous sequence of objects allocated in contiguous memory; all objects are of the same type and there are no gaps between them in memory
- ▶ We can declare an array by writing a declarator of the form `a[d]`, where `a` is the name being introduced and `d` is its size (i.e., the array bound); the size:
 - ▶ specifies the number of elements and must be greater than zero;
 - ▶ is part of the array's type; and
 - ▶ must be known at compile time (must be provide as a constant `expression` or `integer literal`)
- ▶ We can default-initialize an array of built-in type inside a function (such as `main`) by writing

```
int arr[7];
```

- ▶ However, each element will store an undefined value

Notes

Arrays

- ▶ We can explicitly initialize the elements of an array using list notation:

```
int arr[]{0,1,2};
```

- ▶ We can omit the size when we use explicit initialization: the compiler can infer size from the number of initializers
- ▶ Had we provided a size, the number of initializers could not have exceeded that size
- ▶ If the size is greater than the number of initializers provided, the initializers are used for the first elements and any remaining elements are zero-value initialized

Notes

Arrays

- ▶ We can access the elements of an array using subscripting `[idx]`, where `idx` is the index of the element of interest
 - ▶ They are indexed from 0 to `size-1`
- ▶ Arrays are not self-describing:
 - ▶ The number of elements of an array is not guaranteed to be stored within the array
 - ▶ Frequently, a terminator is used to denote whether the end of the array has been reached
 - ▶ Most C++ implementations offer no range checking for arrays

Notes

Arrays and pointers

- ▶ Normally we obtain a pointer to an object using the address-of operator
 - ▶ We can apply the address of operator to any object, including the elements in the array
- ▶ When we use an array, the compiler automatically substitutes a pointer to the first element
- ▶ In fact, the array subscripting operation `arr[idx]` is defined as `*(arr+idx)`
 - ▶ The result of adding an integral value to a pointer is itself a pointer
 - ▶ Therefore, the expression `(arr+idx)` calculates the address `idx` elements from the base of the array
 - ▶ The application of the dereference operator at that address then give you the object at that location in the array

Notes

Overview

Review

Compound types
References
Pointers
Arrays

Compound data
Vectors

References

Notes

Overview

Review

Compound types

References

Pointers

Arrays

Compound data

Vectors

References

Notes

vectors

- ▶ A **vector**, like the array, is a homogeneous sequence of objects allocated in contiguous memory
- ▶ A **vector**:
 - ▶ can hold an arbitrary number of elements up to whatever physical memory and the operating system can handle;
 - ▶ can grow past its declared size using its `push_back()` function;
 - ▶ can start empty and grow as data is added to it using its `push_back()` function;
 - ▶ has range-checking when you access elements using its `at(idx)` function, where `idx` is the index of the element of interest
 - ▶ always knows its own size, which is obtainable through its `size()` function
- ▶ For these reasons and others, prefer **vector** for storing elements over an array (and any other container for that matter) unless you have a good reason not to

Notes

Overview

Review

Compound types

References

Pointers

Arrays

Compound data

Vectors

References

Notes

References

► Lippman, B., Lajoie, Josee, & Moo, B. E. (2016). *C++ primer* (5th ed.). Addison-Wesley.

► Stroustrup, B. (2014). *Programming: principles and practice using C++* (2nd ed.). Addison-Wesley.

Notes

Notes

Notes
