

C/C++ Programming Language

CS205 Spring

Feng Zheng

Week 4



南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY



Content

- Brief Review
- Pointer
- Managing Memory for Data
- Loops and Relational Expressions
- Summary

Brief Review



Compound Types

- Array Types
- Strings
 - C-style String
 - string-class string
- Structure
 - Structure: struct
 - Union: union
 - Enumeration: enum



Pointers



Why Needs a Pointer Type?

- Three fundamental properties of **declaration**

- **Where** the information is stored
- What **value** is kept there - know
- What **type** of information is stored - know

Identity
Student number
Address
Mobile number

- How to know **where** the values are stored?

- Using **address operator** **&** to access the address
- Using **hexadecimal** notation to display the address values
- **Run address.cpp**

✓ // address.cpp -- using the & operator to find addresses

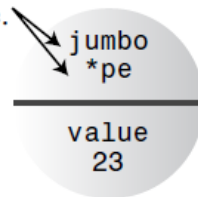


Pointer Type

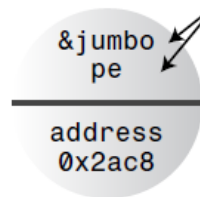
- Using ordinary variables
 - Naturally, the value is treated as a **named** quantity
 - The location as the **derived** quantity
- Using new strategy: **pointer type**
 - **Inverse** way
- Operator of asterisk ***** :
 - Indirect value
 - The dereferencing operator
- Program **pointer.cpp**
 - `// pointer.cpp -- our first pointer variable`

```
int jumbo = 23;  
int * pe = &jumbo;
```

These are
the same.



These are
the same.





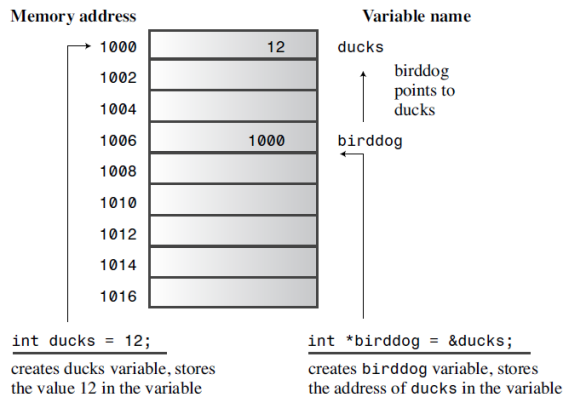
Importance of pointers

- One **essential** to the C/C++ programming philosophy of is the **memory management**
- **Pointers** would be the C/C++ **Philosophy**



Declaring and Initializing Pointers

- Example: `int* birddog;`
 - `* birddog` is a `int` type variable
 - `birddog` is a `pointer` type variable
 - The type for `birddog` is `pointer-to-int`
 - Put the white space **before** or **behind** the `*` or **no** spaces
- `int *` is a compound type
 - `double *`, `float *`, `char *`





Pointer Danger

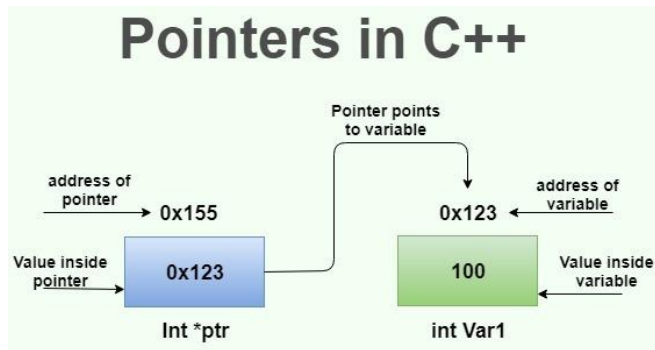
- A confusion for beginners

- Creating a pointer in C++ means the computer allocates memory to **hold an address**

- BUT it **does not** allocate memory to hold the **data**

- ✓ `int * ptr;` // create a pointer-to-int: **NULL**

- ✓ `*ptr= 223323;` // place a value in never-never land: **disaster**





Pointers and Numbers

- Similarities and differences between pointer and integer
 - They are both integers but pointers are not the integer type
 - Both are numbers you can add and subtract but it doesn't make sense to multiply and divide two locations
- Why we need addition and subtraction operations?
- Can't simply assign an integer to a pointer
- You can do like this:
 - 0xB8000000 is an address literal (hexadecimal)
 - `int * ptr = (int *) 0xB8000000;`

Danger!!!



Allocating Memory with **new**

- What is the problem of the pointer? Remember **disaster?**
- How to solve it?
 - The key is the C++ **new** operator
 - ① **Tell** new for what data **type** you want memory
 - ② Let new **find** a block of the correct size
 - ③ **Return** the address of the block
 - ④ **Assign** this address to a pointer
 - ⑤ This is an example: `int * ptr_int = new int; * ptr_int = 1;`
- Now, we have **three** ways of initialization for a pointer type
 - 1. `malloc()`
 - 2. `calloc()`
 - 3. `free()`
 - 4. `realloc()`
- Program `use_new.cpp`
 - Operation: **sizeof**
 - `// use_new.cpp -- using the new operator`



Freeing Memory with **delete**

- **delete** operator enables you to **return** memory to the memory pool
 - The memory can then be **reused** by other parts of the program
 - **Balance** the uses of `new` and `delete`
 - **Memory leak**—memory has been allocated but **no longer** being used
- Beware of
 - Cannot free a block of memory that you have **previously freed**
 - Cannot use `delete` to free memory created by **ordinary variable**

```
int * ps = new int; // allocate memory with new
. . .               // use the memory
delete ps;          // free memory with delete when done
```

```
int * ps = new int; // ok
delete ps;          // ok
delete ps;          // not ok now
int jugs = 5;       // ok
int * pi = &jugs;    // ok
delete pi;          // not allowed, memory not allocated by new
```



Using **new** to Create Dynamic Arrays

- Use **new** with **larger chunks** of data, such as arrays, strings, and structures
 - **Static** binding: the array is built in to the program at **compile time**
 - **Dynamic** binding: the array is created during **runtime**
 - ✓ The **size** of block can be confirm during **runtime**

```
int * psome = new int [10]; // get a block of 10 ints
delete [] psome;           // free a dynamic array
```

- ① Don't use delete to free memory that **new didn't allocate**
- ② Don't use delete to free the same block of memory **twice** in succession
- ③ Use **delete []** if you used **new []** to allocate an **array**
- ④ Use **delete** (no brackets) if you used **new** to allocate a **single** entity
- ⑤ It's safe to apply delete to the **null** pointer (nothing happens)



Using a Dynamic Array

- How do you use the dynamic array?
 - **Identify** every element in the block
 - **Access** one of these elements
- Program `arraynew.cpp`
 - `// arraynew.cpp -- using the new operator for arrays`
 - **A pointer points to the first element**
 - `double * p3 = new double [3]; // space for 3 doubles`
 - `p3 = p3 + 1; // increment the pointer`
 - `p3 = p3 - 1; // point back to beginning`



Pointers, Arrays, and Pointer Arithmetic

- Adding **one** to a pointer variable increases its value by **the number of bytes of the type** to which it points

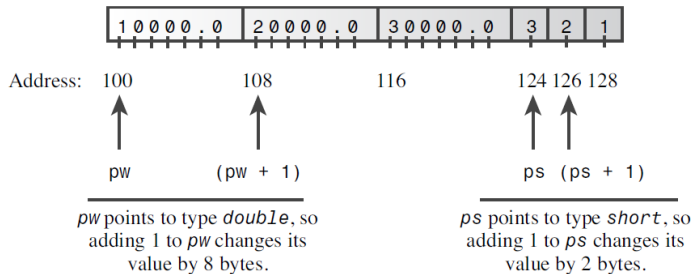
- Program `addpntrs.cpp`

- You can use pointer names and array names in **the same way**

- **Differences** between them

- ① You can **change** the value of a **pointer**, whereas an **array name** is a **constant**
- ② Applying the `sizeof` operator to an array name yields the **size of the array**, but applying `sizeof` to a pointer yields the **size of the pointer**

```
double wages[3] = {10000.0, 20000.0, 30000.0};
short stacks[3] = {3, 2, 1};
double * pw = wages;
short * ps = &stacks[0];
```



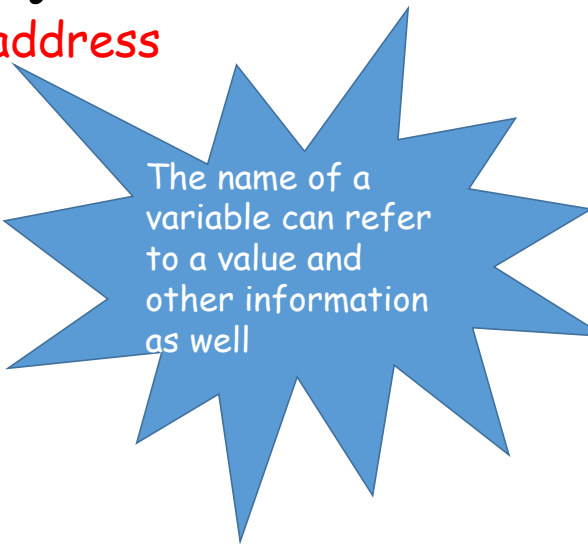


The Address of an Array

type and address

• Program addpntrs-2.cpp

- `short tell[10];`
- `tell` is type **pointer-to-short**
- `&tell` is type **pointer-to-(array of 10 shorts)**
- `short (*pas)[10] = &tell;` // try to replace 10 by 20
- `(*pas) = tell` is type **pointer-to-short**
- `pas=&tell` is type **pointer-to-(array of 10 shorts)**
- `short* pas[10];`
- `pas` is an **array** of 10 pointers-to-short



The name of a variable can refer to a value and other information as well

• **&tell**

- Applying the address operator yields the address of the **whole array**



Summarizing Pointer Points

- Pointers
 - Declaring pointers
 - Assigning values to pointers (three ways)
 - Dereferencing pointers: mean referring to the pointed-to value
 - Distinguishing between a pointer and the pointed-to value
- Array names
 - Bracket array notation is equivalent to dereferencing a pointer
- Pointer arithmetic
- Dynamic binding and static binding for arrays

```
int size;  
cin >> size;  
int * pz = new int [size];    // dynamic binding, size set at run time  
...  
delete [] pz;                // free memory when finished
```



Using **new** to Create Dynamic Structures

- Dynamic means the memory is allocated during **runtime**

- **Creating** the structure
- **Accessing** its members

```
inflatable * ps = new inflatable;
```

- The **arrow membership operator** (->) of a hyphen and then a greater-than symbol

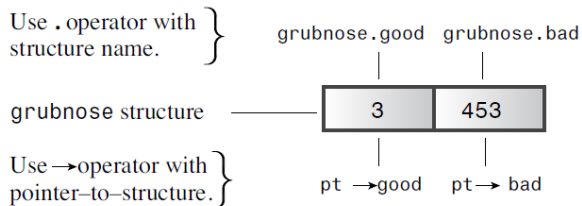
- Program **newstrct.cpp**

```
struct things
{
    int good;
    int bad;
};
```

grubnose is a structure.

```
things grubnose = {3, 453};
things * pt = &grubnose;
```

pt points to the grubnose structure.





An Example of Using **new** and **delete** for Functions

- Program `delete.cpp`

- Return the **address** of the string copy
- It's usually **not** a good idea to put `new` and `delete` in **separate functions**



Address Types

- Pointer

- Address operator: `&`
- Indirect value operator: `*`
- Allocate memory: `new`
- Release memory: `delete`

Managing memory for
data



Important Properties of Variables

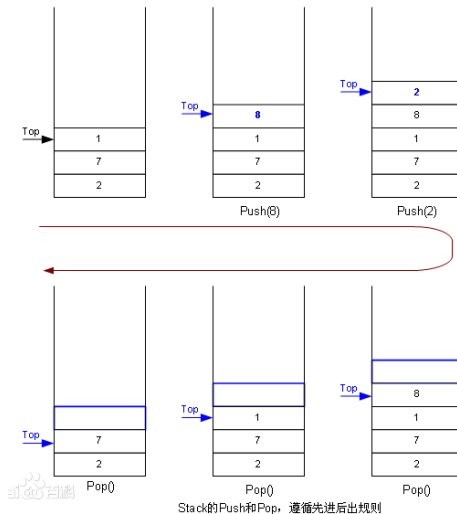
- We have already learned
 - Address
 - Type
 - As for a pointer variable, it contains what type of data it points to
- We need to know
 - Lifetime
 - Scope



Automatic Storage

- Automatic Storage

- Ordinary variables defined inside a function use automatic storage and are called automatic variables
- They expire when the function terminates
- Automatic variables typically are stored on a stack
- A last-in, first-out, or LIFO, process





Static Storage

- Static Storage

- Static storage is storage that exists throughout the execution of **an entire program**
- Two ways
 - ① Define it **externally**, outside a function
 - ✓ To be visible to all files
 - ① Use the keyword **static** when declaring a variable
 - ✓ To define the visible scope: 1) to that file or 2) to certain function

static double fee = 56.50;



Dynamic Storage

- Dynamic Storage

- The **new** and **delete** operators provide a **more flexible** approach than automatic and static variables
- Refer to as the **free store** or **heap**
- Lifetime of the data is **not tied arbitrarily** to the life of the program or the life of a function



Combinations of Types

- Combinations

- Include arrays, structures, and pointers

- Program `mixtypes.cpp`: array of structures

- `const antarctica_years_end * arp[3] = {&s01, &s02, &s03};`
- `const antarctica_years_end ** ppa = arp;`

- Distinguish the following (again)

`type_name * variable_name[10]` ----- `type_name (*variable_name)[10]`



Array Alternatives

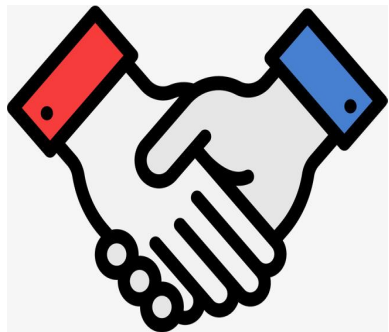
Template??

- The **vector** Template Class
 - It is a **dynamic** array (Similar to the **string** class)
 - Use **new** and **delete** to manage memory
 - The vector identifier is part of the **std** namespace
- The **array** Template Class
 - The **array** identifier is part of the **std** namespace
 - The number of elements **can't** be a **variable**
 - **Static** memory allocation
- See **choices.cpp**
 - Comparing Arrays, Vector Objects, and Array Objects



Summary

- Pointer
- Managing Memory for Data
- Loops and Relational Expressions



Thanks



zhengf@sustech.edu.cn