C++ Program Design
-- C to CPP

Pointers

Junjie Cao @ DLUT

Summer 2022

https://github.com/jjcao-school/c

Pointers

What is a variable?

- a name for a piece of memory that holds a value
- address-of operator (&) allows us to see what memory address is assigned to a variable

```
int main()
{
    int x = 5;
    std::cout << x << '\n'; // print the value of variable x
    std::cout << &x << '\n'; // print the memory address of variable x

    return 0;
}
```

```
the above program printed:
5
0027FEA0
```

The dereference operator (*)

- address-of operator (&)
- dereference operator (*) allows us to get the value at a particular address:

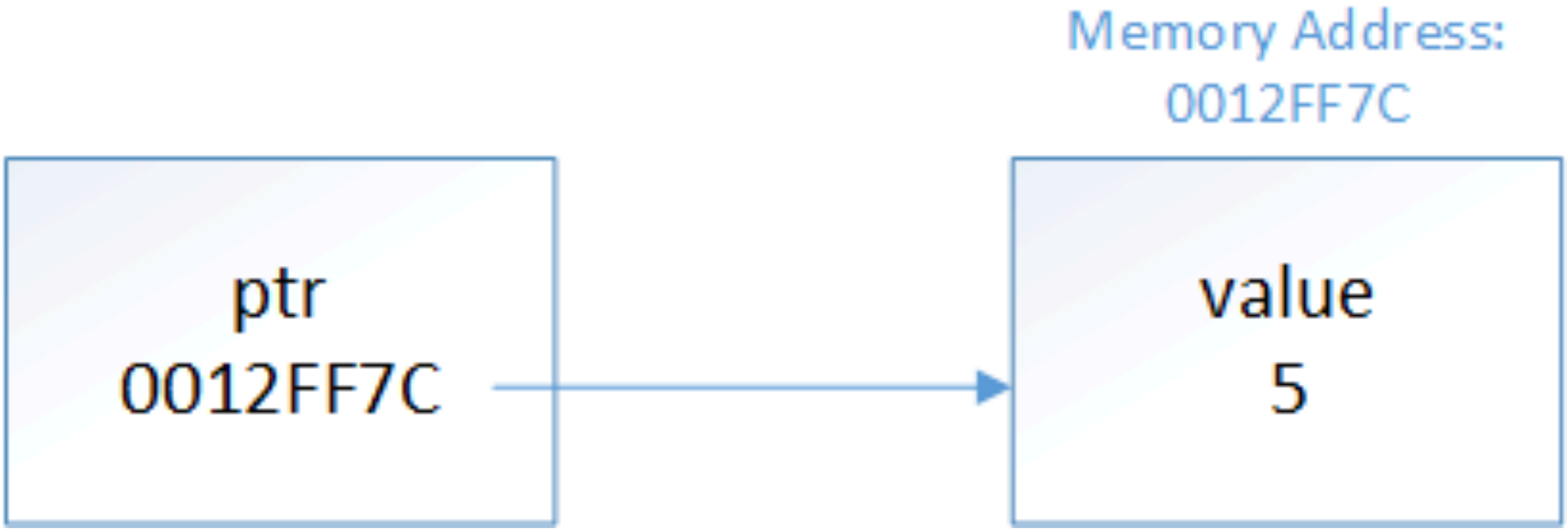
```
int main(){
    int x = 5;
    std::cout << x << '\n'; // print the value of variable x
    std::cout << &x << '\n'; // print the memory address of variable x
    std::cout << *&x << '\n'; // print the value at the memory address of variable x

    return 0;
}
```

- int* iPtr2; // also valid syntax (acceptable, but not favored)
- int * iPtr3; // also valid syntax (but don't do this)
- int* iPtr6, iPtr7; // iPtr6 is a pointer to an int, but iPtr7 is just a plain int!
-
- int *iPtr4, *iPtr5; // declare two pointers to integer variables
- For this reason, when declaring a variable, we recommend putting the asterisk next to the variable name.

- *Best practice: When declaring a function, put the asterisk of a pointer return value next to the type.*
- int* doSomething();

Assigning a value to a pointer

- Since pointers only hold addresses, when we assign a value to a pointer, that value has to be an address
- int value = 5;
- int *ptr = &value; // initialize ptr with address of variable value



```cpp
int main()
{
    int value = 5;
    int *ptr = &value; // initialize ptr with address of variable value

    std::cout << &value << '\n'; // print the address of variable value
    std::cout << ptr << '\n'; // print the address that ptr is holding

    return 0;
}
```

this printed:

0012FF7C
0012FF7C

Change value through pointer

- int value = 5;
- int *ptr = &value; // ptr points to value
- ptr = 7;
- *ptr = 7; // *ptr is the same as value, which is assigned 7
- // prints 7
- std::cout << value; // ?value?

A warning about dereferencing invalid pointers

- Pointers in C++ are inherently **unsafe**, and improper pointer usage is one of the best ways to crash your application.

- When a pointer is dereferenced, the application attempts to go to the memory location that is stored in the pointer and retrieve the contents of memory.

- For security reasons, modern operating systems sandbox applications to prevent them from improperly interacting with other applications, and to protect the stability of the operating system itself.
- If an application tries to **access a memory location not allocated to it by the operating system, the operating system may shut down** the application.

- The following program illustrates this, and will probably crash when you run it (go ahead, try it, you won’t harm your machine):

```
void foo(int *&p){ }


int main(){
    int *p; // Create an uninitialized pointer (that points to garbage)
    foo(p); // Trick compiler into thinking we're going to assign this a valid value

    std::cout << *p; // Dereference the garbage pointer

    return 0;
}
```

The NULL macro & nullptr in C++11

- `int *ptr(NULL); // assign address 0 to ptr`
- NULL is a marco (#define NULL 0) => avoid using it

- *Best practice: With C++11, use keyword* ***nullptr*** *to initialize your pointers to a null value*.
- `int *ptr = nullptr; // note: ptr is still an integer pointer, just set to a null value (0)`

Quiz 1

```
short value = 7; // &value = 0012FF60
short otherValue = 3; // &otherValue = 0012FF54

short *ptr = &value;
*ptr = 9;

std::cout << &value << '\n';
std::cout << value << '\n';
std::cout << ptr << '\n';
std::cout << *ptr  << '\n';
std::cout << '\n';
```

Quiz 2

```
short value = 7; // &value = 0012FF60
short otherValue = 3; // &otherValue = 0012FF54

short *ptr = &value;
*ptr = otherValue;

std::cout << &value << '\n';
std::cout << value << '\n';
std::cout << ptr << '\n';
std::cout << *ptr << '\n';
std::cout << '\n';
```

Quiz 3

```cpp
short value = 7; // &value = 0012FF60
short otherValue = 3; // &otherValue = 0012FF54

short *ptr = &value;
ptr = &otherValue;

std::cout << &value << '\n';
std::cout << value << '\n';
std::cout << ptr << '\n';
std::cout << *ptr << '\n';
std::cout << '\n';
```

Null pointers

- Just like normal variables, pointers are not initialized when they are instantiated.
  - Unless a value is assigned, a pointer will point to some garbage address by default.

```
double *ptr(0);
if (ptr)
    cout << "ptr is pointing to a double value.";
else
    cout << "ptr is a null pointer.";
```

- *Best practice: Initialize your pointers to a null value if you’re not giving them another value.*

Conclusion

- Pointers are variables that hold a memory address.
- They can be dereferenced using the dereference operator (*) to retrieve the value at the address they are holding.
- Dereferencing a garbage pointer may crash your application.

What good are pointers?

- At this point, pointers may seem a little silly, academic, or obtuse. Why use a pointer if we can just use the original variable?
- useful in many different cases:
  - **Arrays** are implemented using pointers.
  - the only way you can **dynamically allocate memory** in C++. the most common use case for pointers.
  - **pass a large amount of data** to a function in a way that doesn't involve copying the data, which is inefficient
  - achieve **polymorphism** when dealing with inheritance
  - have one struct/class point at another struct/class, to form a chain.
    - useful in some more **advanced data structures**, such as linked lists and trees.

Pointers and arrays

Similarities between pointers and fixed arrays

- We know what the values of array[0], array[1], … are 9, 7, …. But what value does array itself have?
- The variable array contains the address of the first element of the array, **as if it were a pointer**!

```
int main(){
    int array[5] = { 9, 7, 5, 3, 1 };
    // print the value of the array variable
    std::cout << "The array has address: " << array << '\n';
    // print address of the array elements
    std::cout << "Element 0 has address: " << &array[0] << '\n';

    return 0;}
```

```
The array has address: 0042FD5C
Element 0 has address: 0042FD5C
```

Differences between pointers and fixed arrays

- an array and a pointer to the array are not identical!
- different type information: int[5] vs. int *
- A fixed array knows how long it is. A pointer to the array does not.

```
int main(){
    int array[5] = { 9, 7, 5, 3, 1 };
    std::cout << sizeof(array) << '\n'; // will print sizeof(int) * array length
    int *ptr = array;
    std::cout << sizeof(ptr) << '\n'; // will print the size of a pointer
    return 0;
}
```

Passing fixed arrays to functions

- copying large arrays can be very expensive, passing pointer instead

```
void printSize(int *array){// array is treated as a pointer here
        std::cout << sizeof(array) << '\n'; // prints the size of a pointer, not the size of the array!
}
int main(){
   int array[] = { 1, 1, 2, 3, 5, 8, 13, 21 };
   std::cout << sizeof(array) << '\n'; // will print sizeof(int) * array length
   printSize(array);

   return 0;
}
```

implicitly conversion

- C++ implicitly converts parameters using the array syntax ([]) to the poi nter syntax (*)
  - => the following two are identical:
    - `void printSize(int array[]);`
    - `void printSize(int *array);`

dynamic memory allocation

The need for dynamic memory allocation

- C++ supports three basic types of memory allocation

  - **Static memory allocation** happens for **static and global** variables. Memory for these types of variables is allocated once when your program is run and persists throughout the life of your program.

  - **Automatic memory allocation** happens for **function parameters** and **local variables**. Memory for these types of variables is allocated when the relevant block is entered, and freed when the block is exited, as many times as necessary.

  - **Dynamic memory allocation**

static and automatic allocation

- Both static and automatic allocation have two things in common:
  - The size of the variable / array must be known at compile time.
  - Memory allocation and deallocation happens automatically (when the variable is instantiated / destroyed).

- If we have to declare the size of everything at compile time, the best we can do is try to **make a guess the maximum size** of variables we'll need and hope that's enough:
- `char name[25]; // let's hope their name is less than 25 chars!`
- `Record record[500]; // let's hope there are less than 500 records!`
- `Monster monster[40]; // 40 monsters maximum`
- `Polygon rendering[30000]; // this 3d rendering better not have more than 30,000 polygons!`

```
char name[25]; // let's hope their name is less than 25 chars!
Monster monster[40]; // 40 monsters maximum
```

- wasted memory

- most normal variables (including fixed arrays) are allocated in a portion of memory called the **stack**.
  - The amount of stack memory for a program is generally quite small
  - VC defaults the stack size to 1MB.

  - If you exceed this number, stack overflow will result, and the operating system will probably close down the program.

Dynamic memory allocation

- `int *ptr = new int; // dynamically allocate an integer and assign the address to ptr so we can access it later`
- `*ptr = 7; // assign value of 7 to allocated memory`

- `int *ptr1 = new int (5); // use direct initialization`
- `int *ptr2 = new int { 6 }; // use uniform initialization`

- `// assume ptr has previously been allocated with operator new`
- `delete ptr; // return the memory pointed to by ptr to the operating system`
- `ptr = 0; // set ptr to be a null pointer (use nullptr instead of 0 in C++11)`

Dangling pointers

- **delete ptr;**

- The delete operator does not *actually* delete anything.
- It simply returns the memory being pointed to back to the operating system.

- The operating system is then free to reassign that memory to another application (or to this application again later).
- Pointers that are pointing to deallocated memory are called **dangling pointer**.

Dangling pointers

```
int main(){
    int *ptr = new int; // dynamically allocate an integer
    *ptr = 7; // put a value in that memory location

    delete ptr; // return the memory to the operating system.  ptr is now a dangling pointer.
    std::cout << *ptr; // Dereferencing a dangling pointer will cause undefined behavior
    delete ptr; // trying to deallocate the memory again will also lead to undefined behavior.

    return 0;
}
```

Dangling pointers

```
int main(){
    int *ptr = new int; // dynamically allocate an integer
    int *otherPtr = ptr; // otherPtr is now pointed at that same memory location
    delete ptr; // return the memory to the operating system.  ptr and otherPtr are now dangling pointers.
    ptr = 0; // ptr is now a nullptr
    // however, otherPtr is still a dangling pointer!
    return 0;
}
```

Rule: To avoid dangling pointers, after deleting memory, set all pointers pointing to the deleted memory to 0 (or nullptr in C++11).

Operator new can fail

- By default, if new fails, a *bad_alloc* exception is thrown.
- If this exception isn't properly handled, the program will simply terminate (crash) with an unhandled exception error.

```
int *value = new (std::nothrow) int; // ask for an integer's worth of memory
if (!value) // handle case where new returned null
{
    std::cout << "Could not allocate memory";
    exit(1);
}
```

Null pointers and dynamic memory allocation

- Null pointers (pointers set to address 0 or nullptr) are particularly useful when dealing with dynamic memory allocation.

```
// If ptr isn't already allocated, allocate it
if (!ptr)
    ptr = new int;
```

- Deleting a null pointer has no effect:

```
if (ptr){
    delete ptr; ptr = 0;}
```

Instead, you can just write:

```
delete ptr;
ptr = 0;
```

Memory leaks

- Dynamically allocated memory effectively has no scope. That is, it stays allocated until it is explicitly deallocated or until the program ends

```
void doSomething(){
    int *ptr = new int;
}
```

- **ptr has no chance to be deleted forever!**
    - ptr is the only variable holding the address
    - ptr will go out of scope.
- This is called a **memory leak**.

Memory leaks

- Memory leaks eat up free memory while the program is running, making less memory available not only to this program, but to other programs as well.

- Programs with severe memory leak problems can eat all the available memory, causing the entire machine to run slowly or even crash.

- int value = 5;
- int *ptr = new int; // allocate memory
- // old address lost, memory leak results
- ptr = &value; //?

- int *ptr = new int;
- // old address lost, memory leak results
- ptr = new int; //?

Dynamically allocating arrays

```
std::cout << "Enter a positive integer: ";
int size;    std::cin >> size;

int *array = new int[size]; // use array new.  Note that size does not need to be constant!

std::cout << "I allocated an array of size " << size << '\n';

array[0] = 5; // set element 0 to value 5

delete[] array; // use array delete to deallocate array
array = 0; // use nullptr instead of 0 in C++11
```

Dynamic arrays are almost identical to fixed arrays

- Array: compiler know its size
- Dynamic array: compiler does not remember its size

Initializing dynamically allocated arrays

- initialize a dynamically allocated array to 0, is simple:
  - `int *array = new int[size]();`

- Prior to C++11, there’s no easy way to initialize it to a non-zero value
  - `int *array = new int[size](5);  //error C3074: an array cannot be initialized with a parenthesized initializer`

- starting with C++11

```
int fixedArray[5] = { 9, 7, 5, 3, 1 }; // initialize a fixed array in C++13
int *array = new int[5] { 9, 7, 5, 3, 1 }; // initialize a dynamic array in C++11
```

下一小节：“const”的用法

“const”关键字的用法

1) 定义常量

const int MAX_VAL = 23 ;

const string SCHOOL_NAME = “Peking University” ;

2) 定义常量指针

◘ 回顾指针

```
int n, m(1);
int * p = & n;
* p = 5;                    // ok
cout << n << endl;  // n= 5
p = &m;   //ok， 指向另外一个变量
cout << n << endl; //n=5
```

2) 定义常量指针

对比指针和引用

```
int n(2), m(1);
int * p = & n;
* p = 5;                // ok
cout << n << endl;  // n= 5
p = &m;   //ok ，指向另外一个变量
cout << n << endl; //n=5
```

```
int n(2), m(1);
int & p = n;
p = 5;
cout << n << endl; //n=5
p = m;//不是指向m，而是改变n的值
cout << n << endl; //n=1
```

2) 定义常量指针

- 不可通过常量指针修改其指向的内容

```
int n=2, m(1);
const int * p = & n;
* p = 5;  //编译出错
n = 4;    //ok
p = &m;  //ok，常量指针的指向可以变化
cout << n << endl  // n= 4
```

2) 定义常量指针

- 不能把常量指针赋值给非常量指针，反过来可以

```
const int * p1; int * p2;
p1 = p2;    //ok
p2 = p1;    //error
p2 = (int * ) p1; //ok,强制类型转换
```

2) 定义常量指针

- 函数参数为常量指针时，可避免函数内部不小心改变参数指针所指地方的内容

```
void MyPrintf( const char * p )
{
    strcpy( p,"this"); //编译出错
    printf("%s",p);    //ok
}
```

```
void MyPrintf( const char & p )
{…}
// 为什么不传普通变量？
//performance
void MyPrintf( const A & a )
{…}
```

3) 定义常引用

- 不能通过常引用修改其引用的变量

```
int n;
const int & r = n;
r = 5; //error
n = 4; //ok
```

comprehensive quiz

- Pointers * and Dereference operator (*)
- new, delete, new [], delete [], & memory leak
- A null pointer is a pointer that is not pointing at anything.
- Reference variable & and Address-of operator (&)

- Pointer to const and const pointer

Quiz time

- What’s wrong with each of these snippets, and how would you fix it?

```
int main(){
   int x = 5;    int y = 7;

   const int *ptr = &x;
   std::cout << *ptr;
   *ptr = 6;
   std::cout << *ptr;
   ptr = &y;
   std::cout << *ptr;

   return 0;}
```

Quiz time

- What’s wrong with each of these snippets, and how would you fix it?

```
int* allocateArray(const int length)
{
    int temp[length];
    return temp;
}
```

Quiz time

- What’s wrong with each of these snippets, and how would you fix it?

```
int main()
{
    double d(5.5);
    int *ptr = &d;
    std::cout << ptr;

    return 0;
}
```