CS31 Discussion

Jie(Jay) Wang

Week8 Nov.18
Outline

• Pointer
• Struct
Memory Management

• When the program gets executed, it gets some amount of memory allocated for use.
Memory Management

• Consider this program

```c
int main() {
    func1(); // call
    func1()
}
void func1() {
    ...
    func2(); // call
    func2()
}
```
Memory Management

- Every variable you create during the program execution gets its own space in some location within the memory. And every location is marked with a unique **address**.

- `int x = 16;`

<table>
<thead>
<tr>
<th>1000</th>
<th>1001</th>
<th>1002</th>
<th>1003</th>
<th>1004</th>
<th>1005</th>
<th>1006</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `char c[] = “cat”`

<table>
<thead>
<tr>
<th>1000</th>
<th>1001</th>
<th>1002</th>
<th>1003</th>
<th>1004</th>
<th>1005</th>
<th>1006</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘c’</td>
<td>‘a’</td>
<td>‘t’</td>
<td>‘\0’</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>1001</td>
<td>1002</td>
<td>1003</td>
<td>1004</td>
<td>1005</td>
<td>1006</td>
</tr>
</tbody>
</table>
Pointers

- **The address-of operator (&):** get the memory address of the expression to the right of the ampersand.

  - `int x = 16;`

    | 1000 | 1001 | 1002 | 1003 | 1004 | 1005 | 1006 |
    |------|------|------|------|------|------|------|
    | 16   |      |      |      |      |      |      |

  - `cout << &x << endl;`
Pointers

• **Pointers** store memory addresses and are assigned a type corresponding to the type of the variable that they point to.

• `<type>* <name>` // declares a pointer of the given `<type>` and calls it `<name>.

  ```
  int* ptrAge;
  bool* ptrValid;
  char* ptrName;
  ```

• To **initialize** pointers

  ```
  int age = 40;
  int* ptrAge = &age;
  ```

  **or**

  ```
  int* ptrAge;
  ptrAge = &age;
  ```

  ![Diagram]

  ```
  int* ptrAge = &age;
  ```
Pointers

- **The dereference operator (*)**: to dereference a pointer to get the variable pointed by the pointer.

```cpp
#include <iostream>
using namespace std;

int main() {
    double x, y;
    // normal double variables
    double *p;
    // a pointer to a double variable
    x = 5.5;
    y = -10.0;
    p = &x;
    // assign x’s memory address to p (make p point to x)
    cout << "p: " << p << endl;
    cout << "*p: " << *p << endl;
    p = &y;
    cout << "p: " << p << endl;
    cout << "*p: " << *p << endl;
    return 0;
}
```
Pointers

• **Question:** Will the code compile? If so, what’s the output?

```cpp
#include <iostream>
using namespace std;
int main(){
    int *ptr;
    cout << *ptr << endl;
}
```

Be careful! Uninitialized pointers can lead to undefined behavior or illegal memory accesses when they haven't been assigned somewhere first. A special keyword `nullptr` that represents “the pointer that points at nothing”.
Pointers

- **Question:** Will the code compile? If so, what’s the output?

```cpp
#include <iostream>
using namespace std;
int main(){
    int *ptr = nullptr;
    cout << *ptr << endl;
}
```

Attempting to dereference a `nullptr` will result in undefined behavior.
Pointers

- We can check to make sure a pointer is or is not null pointer.

```cpp
#include <iostream>
#include <string>
using namespace std;

int main () {
    int i = 50;
    int* latePointer = nullptr;
    if (latePointer == nullptr) {
        latePointer = &i;
    } else {
        cout << "<_<_>_"><endl;
    }
    cout << *latePointer << endl;
}
```
Pointers and Arrays

- int arr[100];
- arr is actually a pointer(int*)
- Special for arr, the pointee can’t change.

- In order to get the value of arr[1]
  - arr[1]
  - *(arr+1)
Pointers and Arrays

• **Question**: Will the code compile? If so, what’s the output?

```cpp
#include <iostream>
using namespace std;

int main(){
    int arr[100];
    int var = 100;
    for (int i = 0; i < 100; i++)
        arr[i] = i;
    cout << *(arr+1) << endl;
    cout << *(&arr[1]) << endl;
    *arr = var;
    cout << arr[0] << endl;
}
```

```
1
1
100
```
Pointers and Arrays

• **Question:** Will the code compile? If so, what’s the output?

```cpp
#include <iostream>
using namespace std;

int main()
{
    int arr[100];
    int var = 100;
    for (int i = 0; i < 100; i++)
        arr[i] = i;
    cout << *(arr+1) << endl;
    cout << *(&arr[1]) << endl;
    arr = &var;
    cout << arr[0] << endl;
}
```

What about `arr[1]`?

`arr+1 = ???

Array elements are located contiguously in memory.
#include <iostream>
using namespace std;

int main(){
    int arr[100];
    int var = 100;
    for (int i = 0; i < 100; i++)
        arr[i] = i;
    cout << arr+1 << endl;
    cout << arr+2 << endl;
}
About `sizeof(int)`

- The only things guaranteed are:
  1. `sizeof(char) == 1`
  2. `sizeof(char) <= sizeof(short)`
  3. `sizeof(short) <= sizeof(int)`
  4. `sizeof(int) <= sizeof(long)`
  5. `sizeof(long) <= sizeof(long long)`
  6. `sizeof(char) * CHAR_BIT >= 8`
  7. `sizeof(short) * CHAR_BIT >= 16`
  8. `sizeof(int) * CHAR_BIT >= 16`
  9. `sizeof(long) * CHAR_BIT >= 32`
  10. `sizeof(long long) * CHAR_BIT >= 64`

- The other things are implementation defined.

http://stackoverflow.com/questions/13398630/why-are-c-int-and-long-types-both-4-bytes
Pointer Arithmetic

• Subtraction and addition to pointers is well defined, such that if I say \((\text{ptr} + i)\), it means "refer to the address \(i\) times \(x\) bytes away from \(\text{ptr}\)," where \(x\) is the size of the type of \(\text{ptr}\).

• Pointers are NOT defined on multiplication or division!

```
#include <iostream>
#include <string>
using namespace std;

int main () {
    double d[] = {1.1, 2.2, 3.3, 4.4, 5.5};
    double* ptr = d; cout << *(ptr * 2) << endl;
}
```
Pointers and Arrays

• You can treat an array variable like a pointer – well, it is a pointer. Therefore, the following are equivalent:

```c
int findFirst(const string a[], int n, string target);
int findFirst(const string* a, int n, string target);
```

• Recall
  • Pass by value
    ```c
    int foo(int n);
    ```
  • Pass by reference
    ```c
    int foo(int &n);
    ```
  • Pass by pointer
    ```c
    int foo(int a[]); int foo(int* a);
    ```
Program Challenge

• The following is one possible implementation of `findFirst()`. Can you modify it such that it doesn’t use brackets?

```c++
int findFirst(const string a[], int n, string target) {
    for (int i = 0; i < n; i++)
        if (a[i] == target)
            return i;
    return -1;
}

int findFirst(const string* a, int n, string target) {
}

cout << findFirst(a, 5, "marge") << endl;
cout << findFirst(a + 2, 3, "marge") << endl;
```
Pointer to pointer

- int** var;

```cpp
#include <iostream>
using namespace std;
int main() {
    int var;
    int *ptr;
    int **pptr;
    var = 3000;
    ptr = &var;
    pptr = &ptr;
    cout << "Value of var = " << var << endl;
    cout << "Value available at *ptr = " << *ptr << endl;
    cout << "Value available at **pptr = " << **pptr << endl;
}
```
Reference to Pointer

- `int* &ptr;`

```cpp
#include <iostream>
using namespace std;

int main() {
    int var1 = 30;
    int var2 = 50;
    int* ptr1 = &var1;
    int* &ptr2 = ptr1;
    cout << *ptr1 << endl;
    ptr2 = &var2;
    cout << *ptr1 << endl;
}
```
Why do we need them?

```cpp
#include <iostream>
int g_n = 42;
void func_ptr(int* pp) {
    pp = &g_n;
}

int main() {
    int n = 23;
    int* pn = &n;
    std::cout << "Before :" << *pn << std::endl;
    func_ptr(pn);
    std::cout << "After :" << *pn << std::endl;
}
```

• Question: how can I get 42 after calling the function?
Solution 1: Pointer to Pointer

```cpp
#include <iostream>

int g_n = 42;
void func_ptr(int** pp) {
    *pp = &g_n;
}

int main() {
    int n = 23;
    int* pn = &n;
    std::cout << "Before :" << *pn << std::endl;
    func_ptr(&pn);
    std::cout << "After :" << *pn << std::endl;
}
```

Before : 23
After : 42
Solution2: Reference to Pointer

```cpp
#include <iostream>

int g_n = 42;

void func_ptr(int* &pp) {
    pp = &g_n;
}

int main() {
    int n = 23;
    int* pn = &n;
    std::cout << "Before:" << *pn << std::endl;
    func_ptr(pn);
    std::cout << "After:" << *pn << std::endl;
}
```

Before: 23
After: 42
Preference of one over the other?

• Now we have seen the syntax of ptr-to-ptr and ref-to-ptr. Are there any advantages of one over the other? I am afraid, no. The usage of one of both, for some programmers are just personal preferences. Some who use ref-to-ptr say the syntax is "cleaner" while some who use ptr-to-ptr, say ptr-to-ptr syntax makes it clearer to those reading what you are doing.

http://www.codeproject.com/Articles/4894/Pointer-to-Pointer-and-Reference-to-Pointer
Programming Challenge

- Define a function, `ptrsToMinMax` that takes in an array of ints, the size of that array, and two int pointers, and then sets each pointer equal to the address holding the minimum and maximum values within that array.

```cpp
#include <iostream>
#include <string>
#include <cassert>
using namespace std;

void ptrsToMinMax (int arr[], int*& min, int*& max, int n) {
    // Simply return if n is 0 or below
    if (n <= 0) {
        return;
    }
    // [!] We'll start off both of our min max pointers at the first element
    min = ???
    max = ???
    // Iterate through all n elements of arr
    for (int i = 0; i < n; i++) {
        // [!] If the current element is less than the min
        // we'll set the min to that element
        if ( ??? ) {
            min = ???
        }
        // [!] If the current element is greater than the max, we'll set the max to that element
        if ( ??? ) {
            max = ???
        }
    }
}

int main () {
    int arr[] = {1, 5, 0, 2, 4};
    int* min; int* max;
    ptrsToMinMax(arr, min, max, 5);
    assert(*min == 0);
    assert(*max == 5);
    cerr << "[!] ALL TESTS PASSED!" << endl;
    return 0;
}
```
Struct

- **Structs** are objects in C++ that represent "data structures", or variables, functions, etc. that are organized under a categorizing identifier.

- **Data Members** are variable components of a given struct; they can be of any variable type.

```cpp
struct <structName> {
    <member1_type> <member1_name>;
    <member2_type> <member2_name>;
    // ...etc.
}; // Remember the semicolon!
```

```cpp
struct Student {
    string name;
    int id;
    string email;
    char grade;
};
```
Struct

```cpp
struct Student {
    string name;
    int id;
    string email;
    char grade;
};
```

- The element/member selection operation (.)

```cpp
const int NUM_STUDENTS = 32;
Student st
Student students[NUM_STUDENTS];
```

```cpp
st.name = "Joe Bruin";
// st’s name is set to “Joe Bruin"
students[10].id = 123456789;
// the 10-th Student in students array is assigned an ID
cout << st.grade << endl;
// print the grade of st
cout << students[0].email << endl;
// print the 0-th Student’s email address
```
Pointers to Structures

• student *p;
• You can refer to a member of the structure pointed by p by first dereferencing it and using the dot operator.
• (*p).name
• Or
• p->name
• This one works only if p is a pointer.
Constructors and Initializing Structs

• A **constructor** is used to initialize the data members of a newly declared struct object; it's how we "initialize" them. Constructors are called automatically at struct instance declaration and are defined by listing a member function with the same name as the struct and NO return type.

  • `StructName () { /* Constructor Body */ }`

• A **parameterized constructor** is simply a constructor that takes parameters when constructing an object.

  • `StructName (paramType1 param1, paramType2 param2, ...) { /* Constructor Body */ }`
Constructors and Initializing Structs

```cpp
Student {
    string name;
    int id;
    string email;
    char grade;
    Student() {
        name = "Jane Doe";
        id = 0;
        email = "aa@a.com";
        grade = 'A';
    }
    Student(string n, int i, string e, char g) {
        name = n;
        id = i;
        email = e;
        grade = g;
    }
};

int main() {
    Student p;
    cout << p.name << endl;
    cout << p.grade << endl;
    Student q("Simon", 12, "simon@ht.com", 'B');
    cout << q.name << endl;
    cout << q.grade << endl;
}
```

Jane Doe
A
Simon
B
Member Functions (Method)

```
#include <iostream>
#include <string>
using namespace std;

struct Student {
    string name;
    int id;
    string email;
    char grade;

    Student() {
        name = "Jane Doe";
        id = 0;
        email = "aa@a.com";
        grade = 'A';
    }

    void printName() {
        cout << name << endl;
    }
};

int main() {
    Student p;
    p.printName();
}
```

What’s the output?

```cpp
#include <iostream>
#include <cstring>
#include <cctype>
#include <string>
using namespace std;
struct trickz {
  int i[5];
  trickz () {
    for (int j = 0; j < 5; j++) {
      // Assume ASCII encoding
      i[j] = j;
    }
  }
};
void trickzInc (trickz* & t, int count) {
  for (int j = 0; j < count; j++) {
    t->i[j]++;
  }
}

int main () {
  trickz t;
  trickz* ptr = &t;
  t.trickzInc(ptr, 5);
  for (int j = 0; j < 5; j++) {
    cout << t.i[j] << endl;
  }
  // True or false?
  cout << (ptr == &t) << endl;
}```
Credit to 2 previous CS31 TAs

• This slide is finished with reference from:
  • Andrew Forney
  • http://web.cs.ucla.edu/~forns/
  • Brian Choi
  • http://netlab.cs.ucla.edu/~schoi/cs31/
Thanks!