Discussion 1E

Jie(Jay) Wang

Week 10 Dec.2
Outline

• Dynamic memory allocation
• Class
• Final Review
Dynamic Allocation of Memory

• Recall...

```c
int len = 100;
double arr[len];  // error!
```

• What if I need to compute the size on-the-fly, and allocate the exact amount of memory for the array?

```c
int len = 100;
double *arr = new double[len];
```

• Syntax

```c
<type>* <name> = new <type>(parameter1, parameter2, ...);
```
Dynamic Allocation of Memory

```c
int *func();
int main()
{
    int *p = func();
    return 0;
}
int *func() {
    int arr[10];
    return arr;
}
```
int *func();
int main()
{
    int *p = func();
    return 0;
}
int *func()
{
    int arr[10];
    return arr;
}
Dynamic Allocation of Memory

```c
int main() {
    int *p = func();
    delete[] p;
    return 0;
}

int* func() {
    int* arr = new int[10];
    return arr;
}
```

Diagram:
- Heap
- arr
- main()
- Some data
int main() {
    int *p = func();
    delete[] p;
    return 0;
}

int* func() {
    int* arr = new int[10];
    return arr;
}
Stack vs. Heap

• **Stack**
  - Local variables, functions, function arguments, etc.
  - Variables in the stack vanish when outside the scope.

• **Heap**
  - Dynamically allocated memory reserved by the programmer
  - Variables in the heap remain until you use delete to explicitly destroy them.
## Stack vs. Heap

<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>What variables live here?</strong></td>
<td>Dynamically allocated memory reserved by the programmer</td>
</tr>
<tr>
<td>Local variables, functions, function arguments, etc.</td>
<td></td>
</tr>
<tr>
<td><strong>How can variables be accessed?</strong></td>
<td>Only through pointers!</td>
</tr>
<tr>
<td>By any type of identifier defined in scope</td>
<td></td>
</tr>
<tr>
<td><strong>Best for storing:</strong></td>
<td>Variables whose size is not known at compile-time</td>
</tr>
<tr>
<td>Local variables that are specific to limited scopes</td>
<td></td>
</tr>
<tr>
<td><strong>Memory is allocated:</strong></td>
<td>Whenever the <code>new</code> keyword is used to initialize a variable and call a constructor</td>
</tr>
<tr>
<td>Whenever a variable is declared in scope</td>
<td></td>
</tr>
<tr>
<td><strong>Memory is freed / deallocated:</strong></td>
<td>Only after the delete keyword is used!</td>
</tr>
<tr>
<td>Whenever a variable disappears from scope (e.g., local variables in a function after returning from that function)</td>
<td></td>
</tr>
</tbody>
</table>

Programmers need to deallocate the memory by themselves!
Memory Leak

• What happens here?

```cpp
int *p;
p = new int[10];
p = new int[8];
```

There is no way to access him anymore...

A simple rule to remember:
*For each new statement, there should be one delete statement.*
Memory Leak

```cpp
int *p = new int;
delete p;
int *p = new int[2];
delete[] p;
int *pArr[10];
delete[] pArr;  
for (int i = 0; i < 10; i++)
    pArr[i] = new int;
```

```cpp
```
```cpp
X
```
Memory Leak

```cpp
int *p = new int;
delete p;

int *p = new int[2];
delete[] p;

int *pArr[10];

for (int i = 0; i < 10; i++)
    pArr[i] = new int;
for (int i = 0; i < 10; i++)
    delete pArr[i];
```
Class

• A **class** is a construct used to group related fields (variables) and methods (functions).

```cpp
class Cat {
    int m_age;
    void meow();
};
```

• One can access these members by using a dot.

```cpp
meowth.m_age    meowth.meow()
```
Class – member functions

• Two ways to define the member functions

```cpp
class Cat {
    int m_age;
    void meow() {
        cout << "Meow~" << endl;
    }
};

Scope resolution operator

void Cat::meow() {
    cout << "Meow~" << endl;
};
```

1. Inside the class definition
2. outside of the class definition
Class – access specifiers

• The output of this code?

```cpp
class Cat {
    int m_age;
    void meow() {
        cout << "Meow~" << endl;
    }
};

int main()
```

The class members have the private access specifier by default and cannot be accessed by an outside class or function.

```cpp
class Cat {
public:
    int m_age;
    void meow() {
        cout << "Meow~" << endl;
    }
};
```

```cpp
int main()
```
Class – access specifiers

meowth.m_age = -3;

• Public members
• Private members

```cpp
int main(){
    Cat meowth;
    meowth.m_age = 3;  // ✗
    meowth.meow();
}
```
Class

• Accessors

```cpp
class Cat {
public:
    int age();
    void meow();
private:
    int m_age;
};
int Cat::age() {
    return m_age;
}
```  

• Modifiers (mutators)

```cpp
class Cat {
public:
    int age();
    void setAge(int newAge);
    void meow();
private:
    int m_age;
};
void Cat::setAge(int newAge) {
    m_age = newAge;
}
```
Class vs. Structs

• Technically, the only difference between a class and a struct is the default access specifier.

• By convention, we use structs to represent simple collections of data and classes to represent complex objects.
  • This comes from C, which has only structs and no member functions.
A **constructor** is a member function that has the same name as the class, no return type, and automatically performs initialization when we declare an object:

```cpp
class Cat {
    public:
        Cat();
        int age();
        void meow();
    private:
        int m_age;
};
Cat::Cat() {
    setAge(0);
    cout << "A cat is born" << endl;
}
```

When a constructor has no arguments, it is called the **default** constructor. The compiler generates an empty one by default.

```cpp
Cat kitty;
// uses default constructor -- Cat()
Cat *p1 = new Cat;
// uses Cat()
Cat *p2 = new Cat();
// uses Cat()
```
Class - Constructors

- We can also create multiple constructors for a class by overloading it.

```cpp
class Cat {
public:
    Cat(); // default constructor
    Cat(int initAge); // constructor with an initial age void meow();
    int age();
    void setAge(int newAge);
private:
    int m_age;
};
Cat::Cat(int initAge) {
    Cat(); // I can call the default constructor,
    setAge(initAge); // and then set the age to initAge
}
Cat kitty1(3);
```
Class - Constructors

• Using initialization lists to initialize fields

```cpp
class Cat {
public:
    Cat();
    int age();
    void setAge(int newAge);
private:
    int m_age;
    int m_weight;
    int m_gender;
};

Cat::Cat(): m_age(0), m_weight(10), m_gender(1)
{ /* some code */ }
```
Class - Constructors

- Dynamic allocation
  
  ```cpp
  Cat *pKitty = new Cat();
  Cat *pKitty2 = new Cat(10);
  pKitty->meow()
  (*pKitty).meow()
  ```
Class - Constructors

• Can this compile? If so, what’s the output?

```cpp
#include<iostream>
using namespace std;
class Cat {
public:
    Cat(int initAge);
    int age();
    void setAge(int newAge);
private:
    int m_age;
};
Cat::Cat(int initAge) {
    setAge(initAge);
}

int main(){
    Cat meowth;
    Cat meowth1(2);
}
```

If we declare a constructor, the compiler will no longer generate a default constructor.
Class - Constructors

• Can this compile? If so, what’s the output?

```cpp
#include<iostream>
using namespace std;
class Cat {
public:
    Cat(int initAge);
    int age();
    void setAge(int newAge);
private:
    int m_age;
};
Cat::Cat(int initAge) {
    setAge(initAge);
}
int Cat::age(){
    return m_age;
}

void Cat::setAge(int newAge){
    m_age = newAge;
}

class Person {
private:
    Cat pet;
};
int main(){
    Person Mary;
}
```
Class - Constructors

• A fixed solution

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

class Cat {
public:
    Cat(int initAge);
    int age();
    void setAge(int newAge);
private:
    int m_age;
};
Cat::Cat(int initAge) {
    setAge(initAge);
}
int Cat::age(){
    return m_age;
}

void Cat::setAge(int newAge){
    m_age = newAge;
}

class Person {
public:
    Person()
        : pet(1){
        cout << "Person initialized" << endl;
    }
private:
    Cat pet;
};

int main(){
    Person Mary;
}
```
Class - Constructors

• Order of construction

• When we instantiate an object, we begin by initializing its member variables *then* by calling its constructor. (Destruction happens the other way round!)

• The member variables are initialized by first consulting the initializer list. Otherwise, we use the default constructor for the member variable as a fallback.

• For this reason, member variable without a default constructor must be initialized through the initializer list.
Class - Constructors

• Can this compile? If so, what’s the output?

class Cat {
    public:
    Cat(string name) {
        cout << "I am a cat: " << name << endl;
        m_name = name;
    }

    private:
    string m_name;
};
class Person {
    public:
    Person(int age) {
        cout << "I am " << age << " years old. ";
        m_cat = Cat("Alfred");
        m_age = age;
    }

    private:
    int m_age;
    Cat m_cat;
};

int main() {
    Person p(21);
}

Person(int age) : m_cat("Alfred") { ... }
A destructor is a special member function of a class that is executed whenever an object of its class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

- The destructor should use delete to eliminate any dynamically allocated variables created by the object.
- A destructor’s name starts with ~, followed by the name of the class, with no return type or arguments.

```cpp
class Cat {
public:
    Cat(int initAge);
    ~Cat();
    int age();
    void setAge(int newAge);
private:
    int m_age;
};
```
Class - this

• This
• The keyword’s value is the address of the object, on which the member function is being called.
• Recall in your project 7

```cpp
bool Arena::addRat(int r, int c) {
    if (!isPosInBounds(r, c))
        return false;

    // Don't add a rat on a spot with a poison pellet
    if (getCellStatus(r, c) != EMPTY)
        return false;

    // Don't add a rat on a spot with a player
    if (m_player != nullptr && m_player->row() == r && m_player->col() == c)
        return false;

    if (m_nRats == MAXRATS)
        return false;

    m_rats[m_nRats] = new Rat(this, r, c);
    m_nRats++;
    return true;
}
```
Final Review

• Refer to UPE Tutoring
  • https://goo.gl/DhAcfN
Week 1

• Compilation/syntax errors
  • Some common syntax errors:
    • Missing semicolons at ends of statements
    • Missing brackets around blocks
    • Missing namespace or #include definitions
    • Misspelled variables or names

• Runtime/logic errors
  • Some common runtime errors:
    • Division by 0
    • Overflow (e.g.: trying to hold a really big number in an int variable that exceeds its bounds)
Strings, chars, loops

• String
• `#include <string>`

<table>
<thead>
<tr>
<th>Operation</th>
<th>What it does</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>string s = &quot;hello&quot;;</code></td>
<td>Declare strings s and s2</td>
<td></td>
</tr>
<tr>
<td><code>string s = &quot;!!!&quot;;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>s.length()</code> or <code>s.size()</code></td>
<td>Return the length of s</td>
<td><code>cout &lt;&lt; s.size(); // prints 5</code></td>
</tr>
<tr>
<td><code>s[i]</code> or <code>s.at[i]</code></td>
<td>Return i-th character. (i should be integer between 0 and size-1 (inclusive))</td>
<td><code>cout &lt;&lt; s[1]; // prints ‘e’</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>cout &lt;&lt; s.at(0); // prints ‘h’</code></td>
</tr>
<tr>
<td><code>s + s2</code></td>
<td>Concatenate two strings</td>
<td><code>cout &lt;&lt; s + s2; // prints “hello!!!”</code></td>
</tr>
</tbody>
</table>
Strings, chars, loops

• Char
• Characters can be represented by a unique integer value.
• ASCII (American Standard Code for Information Interchange) defines the mapping between characters and integers.

<table>
<thead>
<tr>
<th>Operation</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td>char c;</td>
<td>Declare a character c</td>
</tr>
<tr>
<td>isspace(c)</td>
<td>True if c is a whitespace character</td>
</tr>
<tr>
<td>isalpha(c)</td>
<td>True if c is a letter</td>
</tr>
<tr>
<td>isdigit(c)</td>
<td>True if c is a digit</td>
</tr>
<tr>
<td>islower(c)</td>
<td>True is c is a lowercase letter</td>
</tr>
<tr>
<td>isupper(c)</td>
<td>True if c is a uppercase letter</td>
</tr>
</tbody>
</table>

<cctype> library
Strings, chars, loops

- `cin >> var;` command accesses input characters, ignores whitespace, and ignores the newline at the end of the user’s input. We use this to get numerical input, and store it in variable “var”.

- `getline(cin, s);` command consumes all characters up to, and including, the newline character. It then throws away the newline, and stores the resulting string in `s`. We use this to gather string inputs. (requires `<string>` library)

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

int main () {
    string inputString;
    int inputInt;
    cout << "Enter a number: ";
    cin >> inputInt;
    cout << "Input was: " << inputInt << endl;
    cout << "Enter a string: ";
    getline(cin, inputString);
    cout << "Input was: " << inputString << endl;
}
```

Input:
32
world

Output:
Enter a number: 32
Input was: 32
Enter a string: Input was:
CStrings

- C strings are **null-terminated**

```
char s[10] = "HOWAREYOU";
```

<table>
<thead>
<tr>
<th>Operation</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strlen(s)</code></td>
<td>Returns the length of <code>s</code>, not counting ‘\0’.</td>
</tr>
<tr>
<td><code>strcpy(t,s)</code></td>
<td>Copies the string <code>s</code> to <code>t</code>. (Notes: <code>t=s</code> won’t do the job. Also, <code>strcpy</code> doesn’t do the size check for you. You must make sure there’s enough space in <code>t</code> yourself.)</td>
</tr>
<tr>
<td><code>strncpy(t,s,n)</code></td>
<td>Copies the first <code>n</code> characters of <code>s</code> to <code>t</code>. (Note: No size check.)</td>
</tr>
<tr>
<td><code>strcat(t,s)</code></td>
<td>Appends <code>s</code> to the end of <code>t</code>. (Notes: No size check. <code>t += s</code> won’t do the job.)</td>
</tr>
<tr>
<td><code>strcmp(t,s)</code></td>
<td>Compares <code>t</code> and <code>s</code>. Returns 0 if they are equal, something greater than 0 if <code>t &gt; s</code>, and something less than 0 if <code>t &lt; s</code>. (Note: <code>t == s</code> or <code>t &lt; s</code> won’t do the job.)</td>
</tr>
</tbody>
</table>

```c
#include <cstring>
```
Strings, chars, loops

• If-else statements

<table>
<thead>
<tr>
<th>symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>==</td>
<td>Equal to</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Strings, chars, loops

- Switch statements

```c++
switch (expression) {
    case constant1:
        group-of-statements-1;
        break;
    case constant2:
        group-of-statements-2;
        break;
    .
    .
    default:
        default-group-of-statements;
}
```

### Switch example

```c++
switch (x) {
    case 1:
        cout << "x is 1";
        break;
    case 2:
        cout << "x is 2";
        break;
    default:
        cout << "value of x unknown";
}
```

### If-else equivalent

```c++
if (x == 1) {
    cout << "x is 1";
} else if (x == 2) {
    cout << "x is 2";
} else {
    cout << "value of x unknown";
}
```
Strings, chars, loops

- **Loops** let you repeat the same or similar task multiple times. Three primitive loops in C++: while, do-while, and for.

- **while loop**

```cpp
while (condition) {
  body;
}
```

1. Evaluate the condition.
   - If true, run the body.
   - Go back to 1.
   - If false, exit the while loop.

- **do-while loop**

```cpp
do {
  body;
} while (condition);
```

1. Execute the body.
2. Evaluate the condition
   - If true, Go back to 1.
   - If false, exit the while loop.

Notice: The body in do-while loop will be executed once no matter what.
Strings, chars, loops

• for loop

```java
for (initialization; condition; update)
    body;
```

1. Execute initialization.
2. Evaluate the condition.
   If true,
   3. Run the body.
   4. Do the update.
      Go back to 2.
   If false,
   exit the for loop.
functions, reference, scope

• function declaration (function prototype/signature)
• function body (function implementation)
• function call

• A reference variable is an alias, or put simple, another name for an already existing variable.
• `<type>& <name>`
arrays

• Declare an array
  • `<type> <name>[size]`

```c
int a[5] = {1, 2, 3, 5, 7};
int a[] = {1, 2, 3, 5, 7};
```

Out-of-boundary access not allowed!

```c
void fibo10(int fib[]);
void fibo10(int fib[], int n);
```
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>`.

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

• To **initialize** pointers

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

or

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

... 16 ...

1000 1001 1002 1003 1004 1005 1006 ...

int* ptrAge = &age;
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)`
# Pointer Arithmetic

```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+2 << endl;
}
```

arr is pointing to the “integer”. A integer is of 4 bytes on this platform.
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);
    ```
### 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;
}
```

Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
Reference to Pointer

- `int* &ptr;`

```c++
#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;
}
```
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;
};
```
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.
const

- `const int a1 = 3;`
- `const int* a2`
- `int const * a3;`
- `int * const a4`
- `Int const * const a5`

- For member functions in class
  - `int Cat::age() const`
  - `{`
  - `/* code */`
  - `}`
  - Ban age() in class Cat from begging anything which can attempt to alter any member variables in the object.
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!