COMP110: Principles of Computing

Transition to C++ III

Learning outcomes

In this session you will learn how to...

- Define your own **classes** in C++
- Use **pointers**, and allocate objects on the **heap**
- Use **typecasting** to convert values from one type to another
- Use the **CImg** library to write basic GUI applications and image processing algorithms

Object-oriented programming in C++

OOP refresher

- A **class** is a collection of **fields** (data) and **methods** (functions)
- Fields and methods may be **public** (accessible everywhere), **protected** (accessible in the class and classes that inherit from it) or **private** (accessible in the class only)
- Classes may **inherit** fields and methods from other classes
- Subclasses may **override** methods which they inherit — this gives rise to **polymorphism**

Class declarations

```
class MyClass
{
public:
    void doMethod(int x)
    {
        std::cout << x << std::endl;
    }

private:
    int field = 7;
};
```

Fields and methods

- Fields and methods are declared in the class declaration, just like variables and functions
- Class declaration is split into sections by access type (**public**, **protected**, **private**)

Overloading

- Functions and methods can be defined with the **same name** but **different parameters**

```
double getVectorLength(double x, double y)
{
    return sqrt(x * x + y * y);
}

double getVectorLength(Vector v)
{
    return sqrt(v.x * v.x + v.y * v.y);
}
```

Constructors and destructors

```
class MyClass
{
public:
    MyClass()
    {
    }

    ~MyClass()
    {
    }
};
```

- The **constructor** is executed when the class is instantiated
- The **destructor** is executed when the instance is freed

Constructors

- The constructor name matches the class name
- Constructors can take parameters
- The constructor can be overloaded, i.e. can have several constructors with different parameters

```
class MyClass
{
public:
    // Parameterless constructor
    MyClass() { }

    // Constructor with parameters
    MyClass(int x, double y) { }
};
```

Destructors

- The destructor name is the class name prefixed with $\sim$ (tilde)
- Destructors **cannot** take parameters

Modular program design

- Method **declarations** go in the class declaration
- Method **definitions** look like function definitions, with the function name replaced with

  `ClassName::methodName`
- Method definitions can also go inline into the class declaration
  - Best used for short (1 or 2 line) methods
- Good practice: Put class declaration in `ClassName.h`, and method definitions in `ClassName.cpp`

Example: Circle.h

```
#pragma once

class Circle
{
public:
    Circle(double radius);

    double getArea();

private:
    double radius;
};
```

Example: Circle.cpp

```
#include "stdafx.h"
#include "Circle.h"

Circle::Circle(double radius)
    : radius(radius)
{
}

double Circle::getArea()
{
    return M_PI * radius * radius;
}
```

Inheritance

```
class Shape
{
public:
    virtual double getArea();
};

class Circle : Shape
{
public:
    virtual double getArea()
    {
        return M_PI * radius * radius;
    }
};
```

- Methods to be overridden must be marked **virtual**

Pure virtual methods

- **Abstract classes** should never be instantiated — they only exist to serve as a base class
- **Abstract methods** are not defined in the base class, and must be overridden in the subclass
- In C++, abstract methods are called **pure virtual**
- Having at least one pure virtual method **automatically** makes the class abstract

```
class Shape
{
public:
    virtual double getArea() = 0;
};
```

Virtual destructors

- If your class is intended to be subclassed, you should declare the **destructor** to be `virtual`

```
class Shape
{
public:
    virtual ~Shape();
};
```

- Why?
  - http://stackoverflow.com/questions/461203/when-to-use-virtual-destructors
  - http://programmers.stackexchange.com/questions/284561/when-not-to-use-virtual-destructors

Instantiation

- To instantiate with a **parameterless constructor**, just declare a variable

```
MyClass myInstance;
```

- To instantiate with a **constructor with parameters**, add the parameters in parentheses

```
MyClass myInstance(27);
```

- This allocates the instance on the **stack**
- The instance is destroyed (and the destructor is called) when the variable goes **out of scope**

Instantiation: C++ vs Python

```
# Python
myInstance = MyClass()
myOtherInstance = myInstance
```

- `myInstance` is a **reference** to an instance
- `myOtherInstance` is a reference to the **same** instance

```
// C++
MyClass myInstance;
MyClass myOtherInstance = myInstance;
```

- `myInstance` **is** an instance
- `myOtherInstance` is a **different** instance — usually a **copy** of `myInstance` (but it depends on how `MyClass` is defined)

Accessing members

- Use dot (.) notation, similar to Python

```
Circle myCircle(10);
double area = myCircle.getArea();
```

Pointers

Pointers

- A **pointer** is the address of a memory location
- If `T` is a type, `T*` is the type "pointer to `T`"
- `&` is the **address-of** operator: gets a pointer to something
- `*` is the **dereference** operator: gets the thing the pointer points to

Allocating objects on the heap

- Objects can be allocated on the heap using the **`new`** keyword
- **`new`** gives a pointer to the new instance

```
// To use a parameterless constructor
MyClass* myInstance = new MyClass;

// To use a constructor with parameters
MyClass* myOtherInstance = new MyClass(1, 2, 3);
```

Deleting objects from the heap

- Objects instantiated with **`new`** must be deleted using **`delete`**

```
delete myInstance;
```

- Forgetting to do this is a **memory leak**
- Deleting something **twice** is bad
- Trying to **dereference a deleted pointer** is bad

Addressing and dereferencing

```
int a = 7;

// Address-of operator
int* b = &a;

// Dereferencing
int c = *b;
```

- `&` gets the **address** of a variable, i.e. a pointer to it
- `*` **dereferences** the pointer, i.e. looks up the thing it points to

Socrative `6E8NSW3IN`

```
int a = 7;
int* b = &a;
int c = *b;
```

Suppose that the variables are assigned to the following memory addresses:

Variable	a	b	c
Address	1000	1004	1008

1. What is the value of `a`?
2. What is the value of `b`?
3. What is the value of `c`?

Dereferencing for objects

- This would work...

```
Circle* myCircle = new Circle(10);
double area = (*myCircle).getArea();
```

- `->` is a shorthand for dereferencing and accessing a member
- The code below is equivalent to the code above, but clearer

```
Circle* myCircle = new Circle(10);
double area = myCircle->getArea();
```

Null pointer

- Pointers can have a special value **`nullptr`**
- This signifies the pointer doesn't point to anything

```
MyClass* notAnInstance = nullptr;
```

- Similar to `None` in Python
- You may also see `NULL` used instead of **`nullptr`** — the meaning is the same

Polymorphism

- Can have a pointer to a **base class** which is actually an instance of a **derived class**

```
class Shape { ... };
class Circle : Shape { ... };

Shape* myShape = new Circle(10);
std::cout << myShape.getArea() << std::endl;
```

Type conversion

Numeric type conversion

- Most conversions happen automatically on assignment
- Conversions that can “lose” data will give compiler warnings

```
int a = 27;
double b = a;     // OK
double c = 12.3;
int d = c;        // Warning
```

Explicit type conversion

- Common pitfall: an **`int`** divided by an **`int`** is an **`int`**

```
int a = 3;
int b = 2;
double fraction = a / b;
std::cout << fraction << std::endl; // Prints 1.0
```

- The code calculates `a / b` as an **`int`**, then converts the result to a **`double`**
- Need to **cast** the **`int`**s to **`double`**s to get the desired result

```
double fraction = static_cast<double>(a) / static_cast ↩
    <double>(b);
std::cout << fraction << std::endl; // Prints 1.5
```

Static cast

- **static_cast**`<Type>(value)` is for type conversions that the compiler can verify are valid
- E.g. converting between basic (numeric) types
- E.g. converting (pointer to derived class) to (pointer to base class)

```
Circle* myCircle = new Circle(1);
Shape* myShape = static_cast<Shape*>(myCircle);
```

Dynamic casting

- Some casts can fail at runtime
- E.g. converting (pointer to base class) to (pointer to derived class)
- E.g. we have a `Shape*` and want to convert it to a `Circle*`, but what if it's actually a `Square*`?
- `dynamic_cast<Circle*>(myPointer)` will convert the pointer if possible, otherwise it will evaluate to `nullptr`

Other types of cast

- **`reinterpret_cast`**`<>()` reinterprets the bytes in memory as a different type
  - This is dangerous, and only useful in certain specialised circumstances
- **C-style casts** can behave like **`static_cast`** or **`reinterpret_cast`** depending on context
  - Syntax: `(Type)value`
  - Also dangerous, but often used for converting between basic (numeric) types

C-style casts

```
double fraction = static_cast<double>(a) / static_cast ↩
    <double>(b);
```

- You may see this written as

```
double fraction = (double)a / (double)b;
```

- This is more concise, but many C++ programmers consider it bad style

Converting to and from strings

- You **can't** use typecasting to convert values to and from strings
- Instead, use `stringstream`
- There are many examples online

Live coding: Image generation

CImg setup

1. Open Visual C++ 2015 and create a new “Win32 Console Application” (under Templates $\rightarrow$ Visual C++ $\rightarrow$ Win32)
2. Open a web browser to `http://cimg.eu/download.shtml` and download the “Standard Package”
3. Find the `CImg.h` file inside the downloaded zip, and copy it to the project folder created in Step 1 (next to the other `.cpp` and `.h` files)
4. Add the following to the bottom of `stdafx.h`:

```
#include "CImg.h"
using namespace cimg_library;
```