Outline

• A Sample C++ Project
• Basics
• Pointers and Memory Allocation
• Templates
• Inheritance
• References and \texttt{const}
• Other C++ Concepts
Outline

• A Sample C++ Project
• Basics
• Pointers and Memory Allocation
• Templates
• Inheritance
• References and const
• Other C++ Concepts
A Sample C++ Project

Consider a sample lab code for 6.884 you might get.

```
cagnode1:~/6.884/sample_code$ ls
blob.cpp blob.h include lib Makefile test.cpp
```

First, compile the code:

```
cagnode1:~/6.884/sample_code$ make
```

```
g++ -c -o blob.o -g -Wall blob.cpp -I include
g++ -o test.64 -g -Wall test.cpp blob.o -I include
-lpng -L lib/x86_64
```

Then, run the program:

```
cagnode1:~/6.884/sample_code$ ./test.64
Initial force of Q: Force = 0
Final force of Q: Force = 10
```

**Question:** How is this project organized?
Execution begins from the `main` function:

`argc`: # of arguments

`argv`: arguments, as an array of strings

`argv[0]`: program name

`argv[1]` through `argv[argc-1]`: command-line inputs

test.cpp

```cpp
#include "blob.h"
#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {
    Blob Q(100);
    cout << "Initial pos of Q: ";
    Q.print_pos();
    Q.move(10);
    cout << "Final pos of Q: ";
    Q.print_pos();
    return 0;
}
```
The `#include` directive inlines the header file `blob.h` into the current file before `test.cpp` gets compiled.

A header file contains the declarations of objects, functions, etc., that code in other files can use.

For example, `blob.h` contains the declaration of the `Blob` object.
Library header files are included using `<>`.

For example, `iostream` contains the declaration for `cout`, which can be used to print out to the screen.

test.cpp

```cpp
#include "blob.h"
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
    Blob Q(100);
    cout << "Initial pos of Q: ";
    Q.print_pos();

    Q.move(10);
    cout << "Final pos of Q: ";
    Q.print_pos();

    return 0;
}
```
```cpp
#ifndef __BLOB_HEADER__
define __BLOB_HEADER__
#include <png.hpp>
typedef png::rgba_pixel Pixel;

class Blob {
  public:
    Blob(int mass_);
    void move(double new_pos);
    void print_pos();
  
  private:
    int mass;
    double pos;
    Pixel color;
};
#endif // __BLOB_HEADER__
```

begin header guard
type definition
class declaration
constructor
member functions
member variables
need semi-colon!
end header guard
Implementations of a class are often placed in a separate file from the header where the class is declared.

This separation allows programmers to expose the interface of a class without exposing the implementation.
Compilation Process

Preprocessor

- Inlines \#includes, etc.

Compiler

- Translates to machine code.
- Associates calls with functions
- Generates an object file.

Linker

- Associates functions with definitions.

Executable

- test.cpp
- test.o
- blob.o
- test.64
- libc.so
- libpng.a

Libraries

Other object files
Compilation Using g++

Preprocess and compile blob.cpp into an object file:

```g++ -c -o blob.o -g -Wall blob.cpp -I include```

Create an object file. Name of output file. Directories to look for library headers.

Compile test.cpp and link with object files and libraries:

```g++ -o test.64 -g -Wall test.cpp blob.o -I include -lpng -L lib/x86_64```

Load libpng.a Directories to look for libraries.
• A Sample C++ Project
• Basics
• Pointers and Memory Allocation
• Templates
• Inheritance
• References and const
• Other C++ Concepts
Anatomy of a Function

- **Return type**: int
- **Function name**: myfunc
- **Argument list**: (char c, double d)

```c
int myfunc(char c, double d)
{
    // Do stuff
    if ((c == 'a') || (d == 1.0))
        return 4;
    else
        return 12;
}
```

**Forward Declaration**

```c
int myfunc(char c, double d);
```
Anatomy of a Class

- Class name
- Access specifier
- Default Constructor
- Constructor
- Destructor
- Member function
- Access specifier
- Member variables

```cpp
class Thing {
public:
    Thing();
    Thing(char* s);
    ~Thing();
    void f(int i);

private:
    short int v;
    char *s;
};
```
More About Classes

- A `struct` is the same as a `class` except that the default access specifier is public.
- The fully-qualified name of function `f` in class `Thing` (see previous slide) is `Thing::f`
- Within `Thing::f`, `v` and `s` refer to the member variables of the `Thing` being acted upon.
- Within any class member function, `this` is a pointer to the object being acted upon.
- Within `Thing::f`, `this->v` and `this->s` have the same meaning as `v` and `s`, respectively.
Expressions

- Call a global function: 
  \[ x = \text{myfunc('c', 4.2)}; \]
- Call a member function: 
  \[ \text{thingobj.f(x)}; \]
- Arithmetic operators: 
  \[ +, -, *, /, \% \text{ (modulus)} \]
- Increment/decrement 
  \[ ++, -- \text{ (prefix or postfix)} \]
- Logical operators: 
  \[ && \text{(and)}, || \text{(or)}, ! \text{(not)} \]
- Bitwise operators: 
  \[ \&\& \text{(and)}, |\| \text{(or)}, ^\text{(xor)}, \sim \text{(not)}, <<\text{(left-shift)}, >>\text{(right-shift)} \]
- Comparisons: 
  \[ <, >, <=, >=, ==, != \]
- Operate and assign: 
  \[ +=, -=, *=, /=, %=, \\
  &=, |=, ^=, <<=, >>= \]
Control Constructs

- `if (condition) statement else statement`
- `while (condition) statement`
- `do statement while (condition);`
- `for (init; condition; increment) statement`
- `switch (value) {
  case value1: statements
  case value2: statements
  ...
  default: statements
}
- `{ statements... } (compound statement)`
Jump Statements

- **break**; will exit out of the current loop or switch statement.
- **continue**; will immediately begin the next loop iteration
- **return** value; will immediately return from the current function.
- **goto** label; will cause immediate loss of respect for your programming ability.
Variable Declarations

- Declare a variable:
  
  `type name;`
  
  Example: `int x;`

- Initialize a variable:
  
  `type name = expression;`
  
  Example: `int x = 42;`

- Multiple constructor arguments:
  
  `type name(expression);`
  
  Example: `int x(42);`

  `type name(arg1, arg2...);`
  
  Example: `Pixel p(0, 255, 0);`
Data Types


- *user-defined classes and structs*

- A character in single quotes (\'a\') has type **char**.

- A string in double quotes (\"hello\") has type **const char[]** (array of character) but is often passed around as a **const char*** (pointer to character). It is terminated by a null (\'\0\') character.
Console I/O

- Include I/O library: \#include <iostream>
- Print to standard output: std::cout << expr;
- Read from standard input: std::cin >> variable;
- Example:

```cpp
#include <iostream>
int main()
{
    std::cout << "Hello world\n";
    return 0;
}
```
Outline

• A Sample C++ Project
• Basics
• Pointers and Memory Allocation
• Templates
• Inheritance
• References and const
• Other C++ Concepts
Declare a pointer to a char.

Assigns the address of \texttt{c} to \texttt{p}. (& is the \texttt{address} operator.)
char c = 'i';

char *p1;

p1 = &c;

*p1 = 'u';

Declare a pointer to a char.

Assigns the address of c to p. (& is the address operator.)

Change the value of c to 'u' through the pointer p1. (* is the dereference operator.)
Syntactic Sugar: “->”

A simpler way to access `b.mass`.

```c
struct Blob { int mass; };
Blob b;
Blob *pb = &b;
int m1 = (*pb).mass;
int m2 = pb->mass;
```
new can be thought of a function with slightly strange syntax. A call to new:

1. Allocates space to hold the object.
2. Calls the object’s constructor.
3. Returns a pointer to that object.

new can also allocate arrays:

```cpp
int *a = new int[100];
```
Deallocation Using `delete`

```cpp
// Allocate memory
Blob *b = new Blob(100, 10);
int *a = new int[100];
...

// Deallocate memory
delete b;
delete[] a;
```

Every call to `new` should have exactly one corresponding call to `delete`. C++ has no garbage collection.
Destructors

delete on an object b
1. Calls the destructor of b.
2. Frees space occupied by the object.

A destructor cleans up after the object, releasing resources such as memory.

class Edge {
    public:
        Edge();
        virtual ~Edge();
    private:
        Node *m_s, *m_t;
};

Edge::Edge() {
    m_s = new Node(0);
    m_t = new Node(1);
}

Edge::~Edge() {
    if (m_s) delete m_s;
    if (m_t) delete m_t;
}
### Heap vs. Stack Memory

<table>
<thead>
<tr>
<th>On the Heap / Dynamic allocation</th>
<th>On the Stack / Automatic allocation</th>
</tr>
</thead>
</table>

```
Blob* f() {
    Blob *b = new Blob();
    b->move(1);
    ...
    return b;
}
```

The blob object still persists. We can return a pointer and use it later, but we must remember to call `delete`.

```
void g() {
    Blob b;
    b.move(1);
    ...
    return;
}
```

The blob `b` is automatically destroyed when `g()` ends.
Outline

• A Sample C++ Project
• Basics
• Pointers and Memory Allocation
• Templates
• Inheritance
• References and const
• Other C++ Concepts
Templates

- Templates are used for generic programming in C++
- A template is a “proto-class” or “proto-function” used to generate a real class or function at compile time.
- We generate a class from a class template by supplying template parameters.
- We generate a function from a function template by calling it with real arguments.
template <typename T>
class mytmplt {
    public:
    T x;
    int foo(T y);
};

template <class T> has the identical meaning

Different classes instantiated from the same template.

mytmplt<int> var1;
var1.x = 100;
int y1 = 15;
int r1;
r1 = var1.foo(y1);

mytmplt<Point> var2;
var2.x = Point(10, 10);
Point y2(20, 20);
int r2;
r2 = var2.foo(y2);
Function Template Example

Function Template Declaration

```cpp
template <typename A, typename B>
B func (A *a, B b, int c) {
    ...
    return -2*b;
}
```

Using templates

```cpp
Point* p = new Point(10, 20);
int res1; double res2;
res1 = func<Point,int>(p, 10, 5);
res2 = func(p, 1.732, 10);
```

Different functions using the same template.
Template Instantiation

Class Template:
\[
\text{template <typename T>
\class mytmplt}
\]

Function Template:
\[
\text{template <typename A, typename B>
B func (A *a, B b, int c)}
\]

Class:
\[
\class mytmplt<Point>
\]

Function:
\[
\text{double func(Point *a, double b, int c);}
\]

Compile time

Object creation

Object:
\[
\text{var2}
\]

Run time

Function Execution

Call:
\[
\text{func(p, 1.732, 10)}
\]

Adapted from “The C++ Standard Library From Scratch.”
© 2000 Pablo Halpern. Used with permission.
Template Specialization

Suppose we have a generic definition for `mytmplt`.

We can specialize the implementation for particular template parameters.
• STL = Standard Template Library
• Contains well-written, templated implementations of most data structures and algorithms
  ▪ Templates are similar to generics in Java
  ▪ Allows you to easily store anything without writing a container yourself
• Will give you the most hideous compile errors ever if you use them even slightly incorrectly!
```cpp
#include <vector>

std::vector<Point> v;
v.push_back(Point(3, 5));

std::vector<Point>::iterator iter;
for (iter = v.begin(); iter != v.end(); ++iter){
    Point &curPoint = *iter;
    // etc.
}
```

Include library header

Can traverse data structure using iterators.
Outline

• A Sample C++ Project
• Basics
• Pointers and Memory Allocation
• Templates
• Inheritance
• References and const
• Other C++ Concepts
Inheritance

Suppose we have a class for a generic node in a graph.

With inheritance, we can implement a specific kind of graph without reimplementing basic graph functions.

Must include header of base (parent) class.

MyNode inherits from the Node class.
Inherited Variables and Functions

A derived class has access to the protected or public variables/functions of its parent class.

```cpp
void TaskNode::Execute() {
    if (!done) {
        for (int i = 0; i < degree; i++)
            children[i]->Execute();
        Compute();
        done = true;
    }
}
```

For this task graph, the `Compute()` of a node depends on the `Compute()` of all its children.
Outline

• A Sample C++ Project
• Basics
• Pointers and Memory Allocation
• Templates
• Inheritance
• References and const
• Other C++ Concepts
A reference is an alias for another name for an object.

```c++
int x = 10;
int &y = x;

y = 20;
cout << "x is " << x << endl;
cout << "y is " << y << endl;
```

Printed value for x: 20.
Printed value for y: 20.
Pass by Value vs. Pass by Reference

Pass by Value

```cpp
int y = 4;
int z;
z = triple(y);
```

Final value for \(y\): 4.

Final value for \(z\): 12.

Pass by Reference

```cpp
int triple(int &x) {
    x = 3*x;
    return x;
}
```

Final value for \(y\): 12.

Final value for \(z\): 12.
Pass by Reference for Efficiency

```c
struct Blob {
    int x;
    int y;
    int q[100];
};

int foo(Blob &b1, Blob &b2) {
    return b1.x*b2.x + b1.y*b2.y;
}
```

Pass-by-reference is more efficient than pass-by-value for calling `foo(b1, b2)`. For pass by value, the arguments `b1` and `b2` must be copied.
References vs. Pointers

References

```c
int a = 10;
int b = 20;
int &c = a;
c = b;
c = 30;
```

1. References can’t be reseated.
2. Can’t have a reference to nothing.

Pointers

```c
int a = 10;
int b = 20;
int *c = &a;
c = &b;
*c = 30;
```

```c
c = &a;
*c = 40;
c = NULL;
```
When are references useful?

- References (and pointers) can be used to return more than one value: pass multiple parameters by reference.
- Some people find it easier to deal with references rather than pointers, but in the end there is really only a syntactic difference (neither of them pass by value).
- Unless you know what you are doing, do not pass objects by value; either use a pointer or a reference.
Introducing `const`

```plaintext
int y = 4;
int z;
z = triple(y);

int triple(int &x) {
    x = 3*x;
    return x;
}

Final value for `y`: 12.
Final value for `z`: 12.

Won’t compile.
```
Still won’t compile.
**const Pointers**

Pass by `const` reference is extremely common.

**Preferred option**

```cpp
int triple(const int &x) {
    return 3*x;
}
```

For clarity, pass by pointer is preferred to pass by modifiable reference.

```cpp
void triple(int *x) {
    *x = 3*(*x);
}
```
Declaring things \texttt{const}

Read pointer declarations from right to left.

\begin{itemize}
\item \texttt{const River nile;}
  \begin{itemize}
  \item A const River.
  \end{itemize}
\item \texttt{const River* nilePc;}
  \begin{itemize}
  \item (\textit{nilePc} points to a River that is \texttt{const})
  \end{itemize}
\item \texttt{River* const nileCp;}
  \begin{itemize}
  \item (\textit{nileCp} is a \texttt{const} pointer to a River)
  \end{itemize}
\item \texttt{const River* const nileCpc;}
  \begin{itemize}
  \item (\textit{nileCpc} is a \texttt{const} pointer to a \texttt{const} River)
  \end{itemize}
\end{itemize}

© 2010 Pablo Halpern and Jim Sukha
**const References**

Read reference declarations from right to left.

```cpp
const River &nileCr;
(nileCr is a reference to a River that is const)
```

```cpp
River & const nileCr;
(nileCp is a const reference to a River)
```

nileCr can’t be used to change the object.

This is invalid and redundant: references are immutable anyway.

Will this code work?

```cpp
River nile;
const River &nileC;
River &nile1 = nileC;
```
const Methods

A class method that is `const` can not alter member variables of the object.

```cpp
class Point {
public:
    double getX() const;
    double getY() const;
protected:
    double m_x, m_y;
};
```

double Point::getX() const {
    m_x *= 3.0;
    return m_x;
}

Won’t compile.
Returning \texttt{const} References

A \texttt{const} method can return a \texttt{const} reference to a member variable.

\begin{verbatim}
class Point {
 public:
 const double &getX() const;
 const double &getY() const;
 protected:
 double m_x, m_y;
};
\end{verbatim}

\begin{verbatim}
const double&
Point::getX() const
{  return m_x;  }
\end{verbatim}

Valid, since function won’t change \texttt{*this}.
Returning a reference or pointer to a local variable is a **bad idea**.

```cpp
const double &
Point::getZ() const {
    double z = 1.5 * x;
    return z;
}
```

z is allocated on the stack and goes out of scope before getZ() returns!
Outline

• A Sample C++ Project
• Basic Syntax
• Pointers and Memory Allocation
• Templates
• Inheritance
• References and const
• Other C++ Concepts
Some Other C++ Concepts

- Overloading: declaring multiple functions with the same name in the same scope but with different argument types.
- Operator overloading: declaring functions that look like built-in operators.
- Namespaces.
- Conversion constructors and conversion operators.
- More about inheritance: initializer lists, virtual functions, abstract classes, multiple inheritance, etc.
- Exceptions
Call the non–default constructor for the parent class in the initializer list.
Which method gets called?

Which class’s Compute() gets called?

Does anything get printed?

No.
Virtual Functions

```cpp
class TaskNode: public Node {
    virtual void Compute() { // Does nothing };
};
class MyTask: public TaskNode {
    void Compute() { cout << "MyTask"; }
};

MyTask* myTask = CreateMyTaskGraph();
TaskNode *genericTask = (TaskNode*)my_task;

genericTask->Compute();
```

Which class’s Compute() gets called?  **MyTask**

Does anything get printed?  **Yes.**

In Java, functions are “virtual” by default!
More about Virtual Functions

- In Java every method invocation is dynamically bound, meaning for every method invocation the program checks if a subclass has overridden the method. You can disable this (somewhat) by using the keyword `final` in Java.

- In C++ you have to declare the method `virtual` if you want this functionality. (So, “virtual” is the same thing as “not final”)

- Use virtual functions only for object-oriented programming. Generic (template) programming and data-abstraction programming rarely need it.
Some Additional Reading

- **The C++ Standard Library From Scratch**

- **C++ Tutorials**
  [http://www.cprogramming.com/tutorial.html#c++tutorial](http://www.cprogramming.com/tutorial.html#c++tutorial)