

## ACM C++ Tutorial

Nathan Ratliff

## Development Environments

- Windows
  - Microsoft Visual C++
    - Note: It's not ansi standard
- Linux / Unix
  - GCC / G++

## Hello World

```
#include <iostream> // For cout
using namespace std;

int main(){
    cout << "Hello World!" << endl;

    return 0;
}
```

## Compilation Model

- Preprocessor
  - Resolves all preprocessor directives
  - #include, #define macros, #ifdef, etc.
- Compiler
  - Converts text into object files
  - May have unresolved interobject references
- Linker
  - Resolves all interobject references (or gives you a linker error)
  - Creates the binary executable
- Loader
  - Loads the program into RAM and runs the `main()` function

## Compiling with g++

```
tahiti:~$ ls
source_file1.cc  source_file2.cc
tahiti:~$ g++ -c source_file1.cc
tahiti:~$ g++ -c source_file2.cc
tahiti:~$ ls
source_file1.cc  source_file1.o  source_file2.cc
source_file2.o
tahiti:~$ g++ -o exe_prog source_file1.o source_file2.o
tahiti:~$ ls
exe_prog        source_file1.cc  source_file1.o
source_file2.cc source_file2.o
tahiti:~$ ./exe_prog
Hello world!
```

## Simple makefile

```
source_file1.o: source_file1.cc
g++ -c source_file1.cc

source_file2.o: source_file2.cc
g++ -c source_file2.cc

exe_prog: source_file1.o source_file2.o
g++ -o exe_prog source_file1.o source_file2.o
```

## A Cooler Makefile

```
CXX = g++
CXXFLAGS = -Wall -ansi -g

all : exe_prog

source_file1.o : source_file1.cc source_file1.h
$(CXX) $(CXXFLAGS) -c $<

source_file2.o : source_file2.cc source_file2.h
$(CXX) $(CXXFLAGS) -c $<

exe_prog : source_file1.o source_file2.o
$(CXX) -o $@ $^

clean :
rm -f core *.o *~
```

## Preprocessor Directives

- Performed before compilation
  - Compiler errors show up in wierd places for confusing reasons
- `#include <filename>` or `#include "filename"`
  - Copies and pastes filename into the current file
- `#define LABEL [value]`
  - Substitutes value for LABEL everywhere LABEL is used in the body of the code
- IMPORTANT: preprocessor directives are never followed by semicolons!
- `#define` can also be used without specifying a value
  - `#define LABEL`
- We can check for existence such as in

```
#ifdef COW
#define COW_SIZE 12
#endif
int intAry[COW_SIZE];
#else
#define ANIMAL_SIZE
#define ANIMAL_SIZE 1
#endif
int intAry[ANIMAL_SIZE];
#endif
```
- Labels don't have to be capitalized...but everyone's doing it

## #define macros

- `#define` can be used to create macros with parameters

```
#define TAX(x) x * 0.05

int main(){
    int sales = 20;
    // I'm okay :)
    int tax = TAX(sales);
    // I'm evil! :( Why?
    tax = TAX(sales + 10);
    return 0;
}

// Better definition
#define SAFE_TAX(x) (x) * 0.05
```
- Used mostly in C because C++ has templates
- Take a look at <http://lxr.linux.no/source/include/linux/ghash.h>

## Class Syntax

```
class MyClass{
public:
    int getNum(){return m_num;}
    void setNum(int num){m_num = num}
private:
    int m_num;
}; // Don't forget this semicolon!
```

Note: C++ also offers a `struct` aggregated data type, which is very similar to a `class`

## Separating Class Definition from Implementation

```
MyClass.h:
#ifdef MY_CLASS_H
#define MY_CLASS_H

class MyClass{
public:
    int getNum();
    void setNum(int num);
private:
    int m_num;
};

#endif // MY_CLASS_H

MyClass.cc:
#include "MyClass.h"

int MyClass::getNum(){
    return m_num;
}

void MyClass::setNum(int num){
    m_num = num;
}
```

## Procedural Programming

- Writing free floating functions or procedures that are disassociated from all classes
  - C is completely procedural, but C++ mixes both object oriented and procedural programming
- ```
#include <iostream>
using namespace std;

int func1(int i){...}
int func2(int i){...}
int func3(int i){...}
int func4(int i){...}
int func5(int i){...}
int func6(int i){...}

int main(){
    int k =
        func1(func2(func3(func4)));
    cout << func5(func6(k)) << endl;
    return 0;
}
```

## Memory Management

- Pointers
- Memory allocation and deallocation
  - new and delete
  - malloc() and free() (Used mainly in C)
  - Don't mix them!
- Problems and pitfalls
  - Dangling pointers
  - Memory leaks
- Class destructors

## Pointers

- Variables that contain memory addresses
- The myriad uses of \*
  - Declaring pointer `int *i;`
  - Dereferencing `j = *i;`
  - Multiplying `i = j*k;`
- The address of operator &
  - Gets you the address of a variable
    - `int *i = &j;`
      - The pointer i now contains the address of int variable j

## A Pointer Example

```
int main(){
    int i, j;
    int *pi, *pj;

    i = 5;
    j = i;
    pi = &i;
    pj = pi;
    *pj = 4;

    cout << i << " ";
    cout << j << " ";
    cout << *pi << " ";
    cout << *pj << endl;

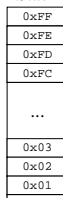
    return 0;
}
```

> 4, 5, 4, 4

## Memory allocation

- Stack vs. heap
  - Stack grows down
  - Heap grows up
- Static Memory
  - Size fixed during compilation
  - Has scope
  - Allocated on stack
- Dynamic Memory
  - Variable size – created at run time
  - Allocated on heap using new
  - Alive until deallocated using delete

Stack



Heap

## Memory Allocation Examples

- new returns a pointer to the dynamically created object.

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

int main(){
    int *i = new int(12);
    Cow *c = new Cow;
    ...
    delete i;
    delete c;

    return 0;
}
```

## Problems

- Dangling pointers
  - Pointers to memory that has already been deallocated
  - segmentation fault (core dump)... or worse....
- Memory leak
  - Loosing pointers to dynamically allocated memory
  - Substantial problem in many commercial products
    - See Windows 98
  - C++ HAS NO GARBAGE COLLECTION!

## Dangling pointer examples

```
int main(){
    int *myNum = new int(12);
    int *myOtherNum = myNum;

    delete myNum;

    cout << *myOtherNum << endl;

    return 0;
}

int* badFunction(){
    int num = 10;
    return &num;
}

int* stillBad(int n){
    n += 12;
    return &n;
}

int main(){
    int num = 12;
    int *myNum = badFunction();
    int *myOtherNum =
        stillBad(num);

    cout << *myNum << ", ";
    cout << *myOtherNum << endl;

    return 0;
}
```

## Memory Leak Examples

```
int main(){
    int *myNum = new int(12);
    myNum = new int(10);
    // Oops...

    delete myNum;

    return 0;
}

int evilFunction(){
    int *i = new int(9);
    return *i;
}

int main(){
    int num = evilFunction();

    // I'm loosing my memory!!

    return 0;
}
```

## Class Destructors

- If a class dynamically allocates memory, we need a way to deallocate it when it's destroyed.
- Destructors called upon destruction of an object

```
class MyClass{
public:
    MyClass(){
        // Constructor
    }
    ~MyClass(){
        // Destructor
    }
    ...
};
```

## Copy Constructor and Assignment Operator

- Copy Constructor:
- Assignment Operator:

```
class Rooster{
public:
    ...
    Rooster(const Rooster &rhs){
        // Do your deep copy
    }
    ...
};
// Usage
Rooster r(12);
Rooster s(r);

class Rooster{
public:
    ...
    Rooster&
operator=(const Rooster &rhs){
    // Copy stuff
}
    ...
};
// Usage
Rooster r(12), s(10);
r = s;
```

## Canonical Form

- All classes should have each of the following:

- Default constructor
- Copy constructor
- Assignment operator
- Destructor

```
// Canonical Cow
class Cow{
public:
    Cow(){...}
    Cow(const Cow &rhs){...}
    Cow& operator=(const Cow &c)
        {...}
    ~Cow(){...}
    ...
};
```

## C++ Arrays

```
#define ARRAY_LENGTH 5

int main(){
    int ary[ARRAY_LENGTH];

    for (int i = 0; i < ARRAY_LENGTH; i++){
        ary[i] = i;
    }

    int *ary2 = ary + ARRAY_LENGTH/2; // Huh?!?
    for (int i = 0; i < ARRAY_LENGTH; i++){
        cout << ary[i] << " ";
    }

    cout << endl;
    for (int i = -ARRAY_LENGTH/2; i < ARRAY_LENGTH/2;i++){
        cout << ary2[i] << " "; // Huh, again...
    }

    cout << endl;
    return 0;
}
```

Output:  
 > 0 1 2 3 4  
 > 0 1 2 3 4

## Arrays Explained

- The name is a label for the address of the first element
  - Array `ary[i]` gives you the value stored in the `i`th memory location after `ary` (in int sized increments)
  - BUT IT IS NOT A POINTER!
    - There is no memory allocated to store the address of the first element
    - Be careful! Interchanging pointers with arrays can get really confusing if you don't understand this.
- The length of a statically allocated array must be known during compilation, i.e. it must be a constant
- If it's not known during compilation, we should use dynamically allocated arrays

## Dynamically Allocated Arrays

- Used when the array length is not known during compilation
- Example:

```
int size;
cin >> size;
int ary[] = new int[size];

for (int i = 0; i < size; i++){
    ary[i] = i;
}

delete [] ary;
```
- Note that the `delete` operator must be prepended with `[]` when used on dynamically allocated arrays

## References

- Lippman, Stanley B. [C++ Primer, Third Edition](#). Addison-Wesely, 1998
- Parlante, Nick. [Pointers and Memory](http://cslibrary.stanford.edu/102/PointersAndMemory.pdf), copyright 1998-2000 <http://cslibrary.stanford.edu/102/PointersAndMemory.pdf>
- Shtern, Victor. [Core C++: A Software Engineering Approach](#). Prentice Hall PTR, 1998
- Stroustrup, Bjarne. [The C++ Programming Language](#). Addison-Wesley, 1997
- ACM Tutorials: <http://www.cs.washington.edu/orgs/acm/tutorials/>
- ACM library in Sieg 326