600.120 Intermediate Programming, Spring 2017*

Misha Kazhdan

**Much of the code in these examples is not commented because it would otherwise not fit on the slides. This is bad coding practice in general and you should not follow my lead on this.*

Outline

- C++ introduction
  - I/O
  - Strings
- Classes
- Constructors, destructors

C++

- C++ is designed to enrich C by proving additional functionality:
  - Classes and inheritance
  - Overloading
  - Templates

- It is not quite a superset of C. Most C programs don't work "out of the box" as C++.

main.cpp

```
#include <iostream>

int main( void )
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

```
>> ./a.out
Hello World!
>>
```

C++

- Stages of compilation are the same as for C:

  preprocess → compile → link → execute

- Use `g++` instead of `gcc`
- Use `-std=c++11` instead of `-std=c99`
- Files end with `.cpp` instead of `.c`

main.cpp

```
#include <iostream>

int main( void )
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

```
>> ./a.out
Hello World!
>>
```

C++

- Our favorite tools work just as well with C++ as with C:
  - `git`
  - `make`
  - `gdb`
  - `valgrind`

main.cpp

```
#include <iostream>

int main( void )
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

```
>> ./a.out
Hello World!
>>
```

C++

#include <iostream>

- As in C, headers provided by C++ are included with angle brackets

- For C++ headers, do not use a trailing `.h`:
  <iostream.h> → <iostream>

- User-defined headers still go in quotes:
  #include "linkedList.h"

main.cpp

```
#include <iostream>

int main( void )
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

```
>> ./a.out
Hello World!
>>
```

C++

- Can use familiar C headers: **assert.h, math.h, ctype.h, stdlib.h, . . .**

main.cpp

```
#include <iostream>
#include <cassert>
int main( int argc , char *argv[] )
{
    assert( argc>1 );
    std::cout << "Hello " << argv[1] << "!" << std::endl;
    return 0;
}
```

```
>> g++ -c main.cpp -std=c++11 -pedantic -Wall -Wextra
>> g++ -o main main.o
>> ./main misha
Hello misha!
>>
```

C++

- Can use familiar C headers: **assert.h, math.h, ctype.h, stdlib.h, . . .**
  - When **#include**'ing, drop **.h** and add **c** at the beginning

main.cpp

```
#include <iostream>
#include <cassert>
int main( int argc , char *argv[] )
{
    assert( argc>1 );
    std::cout << "Hello " << argv[1] << "!" << std::endl;
    return 0;
}
```

```
>> g++ -c main.cpp -std=c++11 -pedantic -Wall -Wextra
>> g++ -o main main.o
>> ./main misha
Hello misha!
>>
```

C++

- Can use familiar C headers: **assert.h, math.h, ctype.h, stdlib.h, . . .**
  - When **#include**'ing, drop **.h** and add **c** at the beginning
  - **argc** and **argv** work as before

main.cpp

```
#include <iostream>
#include <cassert>
int main( int argc , char *argv[] )
{
    assert( argc>1 );
    std::cout << "Hello " << argv[1] << "!" << std::endl;
    return 0;
}
```

```
>> g++ -c main.cpp -std=c++11 -pedantic -Wall -Wextra
>> g++ -o main main.o
>> ./main misha
Hello misha!
>>
```

C++ Input/Output

#include <iostream>

- This is the main C++ library for input and output

main.cpp

```
#include <iostream>

int main( void )
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

```
>> ./a.out
Hello World!
>>
```

C++ I/Output

```
std::cout << "Hello World!" << std::endl;
```

main.cpp

```
#include <iostream>

int main( void )
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

```
>> ./a.out
Hello World!
>>
```

C++ I/Output

<u>std::cout</u> << "Hello World!" << std::endl;

- **std::cout** is the standard output stream
  - Like **stdout** in C

main.cpp

```
#include <iostream>

int main( void )
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

```
>> ./a.out
Hello World!
>>
```

C++ I/Output

```
std::cout << "Hello World!" << std::endl;
```

- **std::endl** is the end-of-line character
  - In C, we called it **'\n'**
  - In C++ it's better to use **std::endl**
    (This flushes the buffer)

main.cpp

```
#include <iostream>

int main( void )
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

```
>> ./a.out
Hello World!
>>
```

C++ I/Output

```
std::cout << "Hello World!" << std::endl;
```

- << is the output-to-stream operator
  - It is processed left to right
  - It returns the output stream

  ⇒ We can chain outputs

main.cpp

```
#include <iostream>

int main( void )
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

```
>> ./a.out
Hello World!
>>
```

C++ I/Output

std::cout << "Hello World!" << std::endl;

main.cpp

```
#include <iostream>
int main(void)
{
    int inventory = 44;
    double price = 0.70;
    const char *item = "chainsaw";
    std::cout << "We have " << inventory << " " << item << "s left,"
        << " costing $" << price << " per unit" << std::endl;
    return 0;
}
```

```
>> ./a.out
We have 44 chainsaws left, costing $0.7 per unit
>>
```

C++ I/Output

```
std::cout << "Hello World!" << std::endl;
```

- No format strings
- Items to print can be chained

```
>> ./a.out
We have 44 chainsaws left, costing $0.7 per unit
>>
```

C++ I/Output

```
std::cout << "Hello World!" << std::endl;
```

- An example of C++ I/O but also an example of *operator overloading*[*]
  - << usually does bitwise left-shift
    If the left operand is a C++ stream (`std::cout`), then << is the output operator

[*]More on this later.

C++ I/Output

Q: How much of C can we use in C++?

A: Nearly everything

main.cpp

```
#include <cstdio>
int main( void )
{
    int inventory = 44;
    double price = 0.70;
    const char *item = "chainsaw";
    printf( "We have %d %ss left costing $%f per unit\n", inventory , item , price );
    return 0;
}
```

```
>> ./a.out
We have 44 chainsaws left costing $0.700000 per unit
>>
```

C++ Input/Output

- Read in from a stream using the >> operator

main.cpp

```
#include <iostream>
#include <string>
int main( void )
{
    std::string first , last;
    int age;
    std::cout << "Please enter your name and age: ";
    std::cin >> first >> last >> age;
    std::cout << last << ", " << first << ": " << age << std::endl;
    return 0;
}
```

```
>> echo misha Kazhdan 25 | ./a.out
Please enter your name and age: kazhdan, misha: 25
>>
```

C++ Input/Output

- Read in from a stream using the >> operator
  - Reads one whitespace-delimited token/word from standard input and places the result in the RHS
  - Returns the input stream

  ⇒ We can chain inputs

main.cpp

```
#include <iostream>
#include <string>
int main( void )
{
    std::string first , last;
    int age;
    std::cout << "Please enter your name and age: ";
    std::cin >> first >> last >> age;
    std::cout << last << ", " << first << ": " << age << std::endl;
    return 0;
}
```

```
>> echo misha Kazhdan 25 | ./a.out
Please enter your name and age: kazhdan, misha: 25
>>
```

C++ Input/Output

- Read in from a stream using the >> operator
  - Reads one whitespace-delimited token/word from standard input and places the result in the RHS
  - Returns the input stream
  - Input stream evaluates to true if it is in a "good" state (no error, no EOF)

main.cpp

```
#include <iostream>
#include <string>
int main( void )
{
    std::string word , earliest;
    while( std::cin >> word )
        if( earliest.empty() || word<earliest )
            earliest = word;
    std::cout << earliest << std::endl;
    return 0;
}
```

```
>> echo "the quick brown fox" | ./a.out
brown
>>
```

C++ Input/Output

- Read in from a stream using the **get** method
  - Reads one whitespace-delimited token/word from standard input and places the result in the RHS
  - Returns the input stream
  - Input stream evaluates to true if it is in a "good" state (no error, no EOF)

main.cpp

```
#include <iostream>
#include <cctype>
int main( void )
{
    char ch;
    while( std::cin.get(ch) )
    {
        ch = toupper(ch);
        std::cout << ch;
    }
    return 0;
}
```

```
>> echo "the quick brown fox" | ./a.out
THE QUICK BROWN FOX
>>
```

*More on this later.

C++ Strings

```
#include <string>
```

- C++ strings provide user-friendly support
- Spare us most of the “nitty-gritty” of C strings
  We still use C strings sometimes (e.g. `char *argv[]`)

C++ Strings

Q: How long can a `std::string` be?

A: Arbitrarily long

Q: Who worries about the memory?

A: C++ library does

- “Backing” memory is dynamically allocated and adjusted as needed
- When `std::string` goes out of scope, associated memory is freed

C++ Strings

- Initialization:
  - `std::string s1 = "world";`
    - initializes to "world"
  - `std::string s2( "hello" );`
    - just like `s2 = "hello"`
  - `std::string s3( 3 , 'a' );`
    - `s2` is "aaa"
  - `std::string s4;`
    - `s4` is the empty string ""
  - `std::string s5( s2 );`
    - copies `s2` into `s5`

C++ Strings

- Operators:
  - `s = "wow";`
    - assign literal to string
  - `std::cin >> s;`
    - put one whitespace-delimited input word in `s`
  - `std::cout << s;`
    - write `s` to standard out
  - `std::getline( is , s );`
    - read to end of line from input stream `is`, store in `s`
  - `s1 = s2;`
    - copy contents of `s2` into `s1`
  - `s1 + s2;`
    - return the string that is `s1` concatenated with `s2`
  - `s1 += s2;`
    - same as `s1 = s1 + s2`
  - `== != < > <= >=`
    - relational operators, using alphabetical ordering

C++ Strings

- Methods:
  - length( );
    - The length of the string
  - capacity( );
    - The maximum string length that can be represented without (internal) reallocation
  - substr(start, size)
    - The substring starting at **start** with length **size**
  - c_str( );
    - A **const char*** version of the string

main.cpp

```
#include <iostream>
#include <cstring>
int main( void )
{
    std::string s = "hello";
    std::cout << s.length( ) << std::endl;
    std::cout << s.capacity( ) << std::endl;
    std::cout << s.substr( 1 , 3 ) << std::endl;
    std::cout << strlen( s.c_str() ) << std::endl;
    return 0;
}
```

```
>> ./a.out
5
15
ell
5
>>
```

C++ Strings

- Accessing:
  - s[5];
    - Gets the 6th character

main.cpp

```
#include <iostream>
#include <string>
int main( void )
{
    std::string s( "Nobody's perfect" );
    for( size_t pos=0 ; pos<=s.length() ; pos++ )
        std::cout << s[ pos ] << " ";
    std::cout << std::endl;
    return 0;
}
```

```
>> ./a.out
N o b o d y ' s   p e r f e c t
>>
```

C++ Strings

- Accessing:
  - s[5];
    - Gets the 6th character
  - s.at(5);
    - Gets the 6th character but first checks that the memory access is in bounds

main.cpp

```
#include <iostream>
#include <string>
int main( void )
{
    std::string s( "Nobody's perfect" );
    for( size_t pos=0 ; pos<=s.length() ; pos++ )
        std::cout << s.at( pos ) << " ";
    std::cout << std::endl;
    return 0;
}
```

```
>> ./a.out
terminate called after throwing an instance of 'std::out_of_range'
  what():  basic_string::at: __n (which is 16) >= this->size() (which is 16)
N o b o d y ' s   p e r f e c t Abort (core dumped)
>>
```

C++ Strings

- See C++ reference for more string functionality
  - www.cplusplus.com/reference/string/string/
- Don't miss:
  - `clear` – set to empty string
  - `append` – just like `+=`
  - `push_back` – like `append` for a single character
  - `insert` – insert one string in middle of another
  - `erase` – remove stretch of characters from string
  - `replace` – replace a substring with a given string

Outline

- C++ introduction
- Classes
- Constructors, destructors

C++ Object Oriented Programming

- We already saw **structs**, which bring together several variables (members) that collectively describe one "thing":

```
…
typedef struct
{
      double w , h;
} Rectangle;
…
```

- We might additionally define some functions (methods) that do things with rectangles, like print them or calculate their area:

```
…
void print( Rectangle r) { std::cout << "width=" << r.w << ", height=" << r.h << std::endl; }
double area( Rectangle r ) { return r.w * r.h; }
…
```

C++ Object Oriented Programming

- We prefer to have the related methods (**print**, **area**) be *part of the object* (the **struct** in this case)
- No simple way to do this in C. But in C++. . .

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
public:
	double w , h;
	void print( void ) const { std::cout << "width=" << w << ", height=" << h << std::endl; }
	double area( void ) const { return w * h; }
};
#endif // RECTANGLE_INCLUDED
```

C++ Object Oriented Programming

- We prefer to have the related methods (**print**, **area**) be *part of the object* (the **struct** in this case)
- No simple way to do this in C. But in C++. . .

rectang

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
public:
	double w , h;
	void print( void ) const { std::cout << "wid
	double area( void ) const { return w * h; }
};
#endif // RECTANGLE_INCLUDED
```

r

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
	Rectangle r = { 30.0 , 40.0 };
	r.print();
	std::cout << "area=" << r.area() << std::endl;
	return 0;
}
```

```
>> ./a.out
width=30, height=40
area=1200
>>
```

C++ Object Oriented Programming

Note:

- The method is called on the object

rectang

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
public:
	double w , h;
	void print( void ) const { std::cout << "wid
	double area( void ) const { return w * h; }
};
#endif // RECTANGLE_INCLUDED
```

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
	Rectangle r = { 30.0 , 40.0 };
	r.print();
	std::cout << "area=" << r.area() << std::endl;
	return 0;
}
```

C++ Object Oriented Programming

Note:

- The method is called on the object
  - The **Rectangle** object is implicit -- it's the one whose **print /area** method is called
  - Methods have access to the members of the calling object

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
public:
    double w , h;
    void print( void ) const { std::cout << "width=" << w << ", height=" << h << std::endl; }
    double area( void ) const { return w * h; }
};
#endif // RECTANGLE_INCLUDED
```

C++ Object Oriented Programming

Note:

- The method is called on the object
- These methods are declared **const** -- calling them won't change the state

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
public:
	double w , h;
	void print( void ) const { std::cout << "width=" << w << ", height=" << h << std::endl; }
	double area( void ) const { return w * h; }
};
#endif // RECTANGLE_INCLUDED
```

C++ Classes

Writing C++ Classes:

- Class definition goes in a .h file

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
public:
	double w , h;
	void print( void ) const { std::cout << "width=" << w << ", height=" << h << std::endl; }
	double area( void ) const { return w * h; }
};
#endif // RECTANGLE_INCLUDED
```

C++ Classes

Writing C++ Classes:

- Class definition goes in a `.h` file
- The definition is preceded by the keyword **class**
- Like **struct**s the definition is closed with a ";"

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
public:
    double w , h;
    void print( void ) const { std::cout << "width=" << w << ", height=" << h << std::endl; }
    double area( void ) const { return w * h; }
};
#endif // RECTANGLE_INCLUDED
```

C++ Classes

Writing C++ Classes:

- Can define a method in the class definition (if it's *very* short)

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
public:
	double w , h;
	void print( void ) const { std::cout << "width=" << w << ", height=" << h << std::endl; }
	double area( void ) const { return w * h; }
};
#endif // RECTANGLE_INCLUDED
```

C++ Classes

Writing C++ Classes:

- Otherwise, declare them in the .h file and define them in a .cpp file

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
public:
	double w , h;
	void print( void ) const;
	double area( void ) const ;
};
#endif // RECTANGLE_INCLUDED
```

rectangle.cpp

```
#include <iostream>
#include "rectangle.h"
void Rectangle::print( void ) const { std::cout << "width=" << w << ", height=" << h << std::endl; }
double Rectangle::area( void ) const { return w * h; }
```

C++ Classes

Writing C++ Classes:

- Otherwise, declare them in the .h file and define them in a .cpp file
  - The method name is preceded by the name of the class and "::" to indicate which class the method belongs to

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
public:
	double w , h;
	void print( void ) const;
	double area( void ) const ;
};
#endif // RECTANGLE_INCLUDED
```

rectangle.cpp

```
#include <iostream>
#include "rectangle.h"
void Rectangle::print( void ) const { std::cout << "width=" << w << ", height=" << h << std::endl; }
double Rectangle::area( void ) const { return w * h; }
```

C++ Classes

Writing C++ Classes:

- Members / methods can be **public** or **private** (or **protected**, discussed later)

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
public:
	double w , h;
	void print( void ) const;
	double area( void ) const ;
};
#endif // RECTANGLE_INCLUDED
```

C++ Classes

Writing C++ Classes:

- Members / methods can be **public** or **private** (or **protected**, discussed later)
- We use **public:** and **private:** to divide the class definition into sections according to whether members are public or private
- Everything is **private** by default
- A **public** member / method can be accessed by any code with access to the class definition (code that includes the **.h** file)
- A **private** member / method can be accessed from other methods in the class, but *not* by a user of the class

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
public:
    double w , h;
    void print( void ) const;
    double area( void ) const ;
};
#endif // RECTANGLE_INCLUDED
```

C++ Classes

Writing C++ Classes:

⇒ We can protect methods / members by making them **private**

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
    double _w , _h;
public:
    void print( void ) const;
    double area( void ) const ;
};
#endif // RECTANGLE_INCLUDED
```

rectangle.cpp

```
#include <iostream>
#include "rectangle.h"
void Rectangle::print( void ) const { std::cout << "width=" << _w << ", height=" << _h << std::endl; }
double Rectangle::area( void ) const { return _w * _h; }
```

C++ Classes

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
    Rectangle r;
    std::cout << r._w << std::endl;
    return 0;
}
```

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
    double _w , _h;
public:
    void print( void ) const;
    double area( void ) const ;
```

```
ECTANGLE_INCLUDED
```

```
>> g++ main.cpp rectangle.cpp -std=c++11 -pedantic -Wall -Wextra
main.cpp: In function int main() :
main.cpp:6:18: error: double Rectangle::_w is private within this context
   std::cout << r._w << std::endl;
                  ^~
...
>>
```

```
void Rectangle::print( void ) const { std::cout << "width=" << _w << ", height=" << _h << std::endl; }
double Rectangle::area( void ) const { return _w * _h; }
```

C++ Classes

Writing C++ Classes:

- We can give read access to the value of a member by defining methods that return a copy

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
    double _w , _h;
public:
    void print( void ) const;
    double area( void ) const ;
    double width( void ) const { return _w; }
    double height( void ) const { return _h; }

ndif // RECTANGLE_INCLUDED
```

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
    Rectangle r;
    std::cout << r.width() << std::endl;
    return 0;
}
```

C++ Classes

Writing C++ Classes:

- We cannot initialize members when they are declared*

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
    double _w=10 , _h=20;
public:
    void print( void ) const;
    double area( void ) const ;
    double width( void ) const { return _w; }
    double height( void ) const { return _h; }

ndif // RECTANGLE_INCLUDED
```

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
    Rectangle r;
    std::cout << r.width() << std::endl;
    return 0;
}
```

*This kind of initialization is only allowed for **static** members

Outline

- C++ introduction
- Classes
- Constructors, destructors

C++ Classes

Q: How do class members get initialized?

A: A *constructor* is code that is called when an object is first created, and can (should) be used to initialized values

C++ Default Constructors

- The *default constructor* is called when no initialization parameters are passed

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
    double _w , _h;
lic:
    void print( void ) const;
    double area( void ) const ;
    double width( void ) const { return _w; }
    double height( void ) const { return _h; }

ndif // RECTANGLE_INCLUDED
```

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
    Rectangle r;     // Default constructor called here
    …
}
```

C++ Default Constructors

- The *default constructor* is called when no initialization parameters are passed
  - If no default constructor is given, C++ implicitly defines one which calls the default constructors of the methods

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
    double _w , _h;
public:
    void print( void ) const;
    double area( void ) const ;
    double width( void ) const { return _w; }
    double height( void ) const { return _h; }
};
#endif // RECTANGLE_INCLUDED
```

C++ Default Constructors

- The *default constructor* is called when no initialization parameters are passed
  - Or the class can provide its own
    - Looks like a function:
      - Whose name is the class name
      - With no (void) arguments
      - With no return type
    - This should be public

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
    double _w , _h;
public:
    Rectangle( void ){ _w = 10 , _h = 20; }
    void print( void ) const;
    double area( void ) const ;
    double width( void ) const { return _w; }
    double height( void ) const { return _h; }
};
#endif // RECTANGLE_INCLUDED
```

C++ Default Constructors

- The *default constructor* is called when no initialization parameters are passed
  - Or the class can provide its own
    - It can be defined in the class definition (if it's short)

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
    double _w , _h;
public:
    Rectangle( void ){ _w = 10 , _h = 20; }
    void print( void ) const;
    double area( void ) const ;
    double width( void ) const { return _w; }
    double height( void ) const { return _h; }
};
#endif // RECTANGLE_INCLUDED
```

C++ Default Constructors

- The *default constructor* is called when no initialization parameters are passed
  - Or the class can provide its own
    - It can be defined in the class definition (if it's short)
    - Or it can be declared in the `.h` file and defined the `.cpp` file

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
    double _w , _h;
public:
    Rectangle( void );
    void print( void ) const;
```

rectangle.cpp

```
#include <iostream>
#include "rectangle.h"
Rectangle::Rectangle( void ){ _w = 10 , _h = 20; }
…
```

C++ Default Constructors

- The *default constructor* is called when no initialization parameters are passed
  - You cannot call the constructor directly.
    - A constructor is called when a new object is declared on the stack or when it is created on the heap using **new***

```
main.cpp
#include "rectangle.h"
int main( void )
{
    Rectangle r;
    Rectangle *rPtr = new Rectangle();
    …
    return 0;
}
```

*We will discuss the **new** operator later.

C++ Default Constructors

Note:

- We've been using default constructors behind the scenes

main.cpp

```
#include <iostream>
#include <string>
int main( void )
{
    std::string name;
    std::cout << "Please enter your first name: ";
    std::cin >> name;
    std::cout << "Hello, " << name << "!" << std::endl;
    return 0;
}
```

C++ Non-Default Constructors

- Constructors can also take arguments, allowing caller to “customize” the object

```
main.cpp
#include <iostream>
#include <string>
int main( void )
{
    std::string s1( "hello" );
    std::string s2 = "goodbye";      // std::string s2 = std::string( "goodbye" );
    std::cout << s1 << " " << s2 << std::endl;
    return 0;
}
```

```
>> ./a.out
hello goodbye
>>
```

C++ Non-Default Constructors

- Constructors can also take arguments, allowing caller to "customize" the object

main.cpp

```
#include <iostream>
#include "rectangle.h"
int main( void )
{
    Rectangle r1 , r2 ( 5 , 5 );
    std::cout << r1.width() << std::endl;
    std::cout << r2.width() << std::endl;
    return 0;
}
```

```
>> ./a.out
10
5
>>
```

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
    double _w , _h;
public:
    Rectangle( void ){ _w = 10 , _h = 20; }
    Rectangle( int w , int h ){ _w=w , _h=h; }
    ...
};
#endif // RECTANGLE_INCLUDED
```

C++ Non-Default Constructors

- Constructors can also take arguments, allowing caller to "customize" the object
  - Note that this implies that we can have two functions with the same name but with different arguments*

rectangle.h

```
#ifndef RECTANGLE_INCLUDED
#define RECTANGLE_INCLUDED
class Rectangle
{
    double _w , _h;
public:
    Rectangle( void ){ _w = 10 , _h = 20; }
    Rectangle( int w , int h ){ _w=w , _h=h; }
    ...
};
#endif // RECTANGLE_INCLUDED
```

*More on function overloading later.

C++ Non-Default Constructors

- Before the body of the constructor is called, C++ calls the default constructor of each of the methods

main.cpp

```
#include <iostream>
#include <string>
class MyString
{
public:
    std::string str;
    MyString( void ){ str = "hello"; }
};
int main( void )
{
    MyString s;
    std::cout << s.str << std::endl;
    return 0;
}
```

```
>> ./a.out
hello
>>
```

C++ Non-Default Constructors

- Before the body of the constructor is called, C++ calls the default constructor of each of the methods
  - This is inefficient because the default constructor of **MyString** undoes the default construction of **str** with the results of a different constructor
  - We would like to be able to invoke the non-default constructor directly

main.cpp

```
#include <iostream>
#include <string>
class MyString
{
public:
    std::string str;
    MyString( void ){ str = "hello"; }
};
int main( void )
{
    MyString s;
    std::cout << s.str << std::endl;
    return 0;
}
```

```
>> ./a.out
hello
>>
```

C++ Non-Default Constructors

- *Initializer lists* allow us to specify that a (non-default) constructor should be used to initialize the member directly
  - Before defining the body of the constructor:
    - a ":" followed by a comma-separated list of member constructors

main.cpp

```
#include <iostream>
#include <string>
class MyString
{
public:
    std::string str;
    MyString( void ) : str( "hello" ) {}
};
int main( void )
{
    MyString s;
    std::cout << s.str << std::endl;
    return 0;
}
```

```
>> ./a.out
hello
>>
```

C++ Non-Default Constructors

- *Initializer lists* allow us to specify that a (non-default) constructor should be used to initialize the member directly
  - Before defining the body of the constructor:
    - a ":" followed by a comma-separated list of member constructors
  - Can do this for primitive types that do not have constructors

main.cpp

```
#include <iostream>
class Foo
{
public:
    int x , y;
    Foo( void ) : x(5) , y(10) {}
};
int main( void )
{
    Foo f;
    std::cout << f.x << " " << f.y << std::endl;
    return 0;
}
```

```
>> ./a.out
5 10
>>
```

C++ Destructors

- A class *constructor*'s job is to initialize the fields of the object
  - It's common for a constructor to obtain a resource (allocate memory, open a file, etc.) that should be released when the object is destroyed

- A class *destructor* is a method called by C++ when the object goes out of scope or is otherwise deallocated (e.g. using `delete`*)

*We will discuss the `delete` operator later.

C++ Destructors

- A class *constructor*'s job is to initialize th
  - It's common for a constructor to obtain a re
    file, etc.) that should be released when the

- A class *destructor* is a method called by
  of scope or is otherwise deallocated (e.g
    - Looks like a function:
      - Whose name is the class name
        - prepended with a "~"
      - With no (void) arguments
      - With no return type
    - This should be **public**

main.cpp

```
#include <iostream>
#include <cassert>
class Foo
{
public:
    size_t sz;
    int* values;
    Foo( int s ) : sz( s )
    {
        values = malloc( sizeof(int)*sz );
        assert( values );
    }
    ~Foo( void ){ free( values ); }
};
int main( void )
{
    Foo f( 10 );
    return 0;
}
```

C++ Destructors

- A class *constructor*'s job is to initialize th
  - It's common for a constructor to obtain a re
    file, etc.) that should be released when the
- A class *destructor* is a method called by
  of scope or is otherwise deallocated (e.g
    - As with other methods, it can be defined in the class definition or outside of it

main.cpp

```
#include <iostream>
#include <cassert>
class Foo
{
public:
    size_t sz;
    int* values;
    Foo( int s ) : sz( s )
    {
        values = malloc( sizeof(int)*sz );
        assert( values );
    }
    ~Foo( void );
};
Foo::~Foo( void ){ free( values ); }
int main( void )
{
    Foo f( 10 );
    return 0;
}
```