

Simple Data Types

1. Integral
 - (a) char examples: 'a' '3' '!'
 - (b) short, int, long examples: 3 -5
 - (c) bool examples: true false 1 0
2. Floating-point
 - (a) float, double examples: -4.1 5.23 3.12E5

String Type

literals: examples: "1984" "I am a house."
 access to class: #include <string>

Constants

declaration: const data-type identifier = value;
 example: const int MAX_LEN = 100;

Variable Declaration

declaration: data-type identifier[, identifier, ...];
 examples: int temp;
 int year;
 double f, c;
 string name;

Arithmetic Operators and Expressions

1. basic operators
 - (a) addition: + example: 3 + 4 result: 7
 - (b) subtraction: - example: 7 - 9 result: -2
 - (c) multiplication: * example: 2.4 * 1.5 result: 3.6
 - (d) division: / example: 7.1 / 2 result: 3.55
 - (e) remainder: % example: 7% 3 result: 1
2. order of operations:
 - (a) highest: * / %
 - (b) lowest: + -
3. parenthesis: () changes the order of operations
 - example: 3 * 2 - 4 * 5 result: -14
 - 3 * (2 - 4) * 5 result: -30

Type Conversion

1. simple: (result-type) value
 example: (int) 3.1 result: 3
2. static: static_cast<result-type>value
 example: static_cast<int> 3.1 result: 3

Assignment

operator: =
 syntax: variable = expression
 examples: (assume int i; double j; string s;)
 i = 35;
 j = 3.4 * b / c;
 s = "bob";
 int f = 72;
 string name = "Bob Cratchett";

Block or Compound Statement

purpose: combines multiple statements into one statement; defines scope of identifiers
 syntax: {
 other statements
 }
 example: int i, j;
 {
 i = 33;
 j = 33 * 4;
 }

String Manipulation

concatenation: string s = "Bob", t = "Cratchett";
 string name = s + " " + t;

Unary Increment Operator

++
 example: int i = 33;
 i++; result: i is 34
 ++i; result: i is 35

Unary Decrement Operator

--
 example: int i = 33;
 i--; result: i is 32;
 --i; result: i is 31;

Output Stream Insertion Operator

<<
 example: << expression
 common escape characters: \n newline
 \t tab
 \\ slash output
 \' single quote output
 \" double quote output
 newline constant: endl

Standard Device Output

library access: #include <iostream>
 standard device var: cout
 some syntax of use: cout << expression [<< expression];

File Output

class: ofstream
 class access: #include <fstream>
 variable declaration: ofstream identifier;
 creating output stream: identifier.open(c-string-path);
 output to file: identifier << expression [<< expression];
 closing output stream: identifier.close();
 example: ofstream outFile;
 outFile.open("data.dat");
 outFile << "This is file output" << endl;
 outFile.close();

Input Stream Extraction Operator

>> reads from stream to white-space
 example: >> variable

Standard Device Input

library declaration: #include <iostream>
 standard device var: cin
 some syntax of use: cin >> variable[>> variable];

File Input

class: ifstream
 class access: #include <fstream>
 variable declaration: ifstream identifier;
 creating input stream: identifier.open(c-string-path);
 input from file: identifier >> expression [>> expression];
 closing input stream: identifier.close();
 example:

```
int a, b;
ifstream inFile;
inFile.open("data.dat");
if (!inFile)
{
    cout << "no file" << endl;
    exit(0);
}
inFile >> a >> b;
cout << a + b << end;
outFile.close();
```

Function getline

syntax: getline(input-variable, string-variable);
 function: reads an entire line, including white-space
 example: string name;

```
cout << "Enter your full name: ";
getline(cin, name);
```

caveat: A getline that follows the use of an input stream operator will read starting from the whitespace that stopped the input stream operator. If the the desired data is on the next line of input, the whitespace must be read in first.

examples:

```
int i; string s;
cin >> i;
cin.get();
getline(cin, s);

int j; string t, dummy;
cin >> t;
getline(cin, dummy);
getline(cin, t);
```

Formating Output

library access: #include <iomanip>
 maipulators: fixed - disables E notion
 showpoint - forces output of .
 functions: setprecision(fraction-size)
 setw(minimum-number-spaces)
 example: double PI = 3.14157;
 cout << fixed << showpoint;
 cout << setPrecision(2) << setw(10)
 << "PI:" << PI << endl;
 example result: PI: 3.14

Relational Operators

== true if the left expression equals the right
 != true if the left expression does not equal the right
 < true if the left expression is less than the right
 > true if the left expression is greater than the right
 <= true if the left expression is less than or equal to the right
 >= true if the left exp. is greater than or equal to the right

Logical Operator

& true if the left and right expression are true, both expressions are evaluated
 && true if the left and right expression are true, both expressions are evaluated only if the left expression is true
 | true if the left or right expression is true, both expressions are evaluated
 || true if the left or right expression is true, both expressions are evaluated only if the left expression is false
 ! not, reverses the value of a boolean expression

If Selection

syntax: if (boolean-expression)
 statement
 if (boolean-expression)
 statement
 else
 statement
 if (boolean-expression)
 statement
 else [if (boolean-expression)
 statement]

examples: if (i > 3)
 j = 33.5;
 if (i>3)
 {
 j = 33.5;
 k = 52.7;
 }

If Selection (continued)

```

if (i > 3)
{
    j = 33.5;
    k = 52.7;
}
else
{
    j = 0;
    k = 0;
}

if (i > 3)
{
    j = 33.5;
    k = 52.7;
}
else if (i > 6)
{
    j = -45.0;
    k = -23.9;
}
else
{
    j = 0;
    k = 0;
}
    
```

Switch Selection

```

typical syntax: switch(expression)
{
case value1:
    statement1
    break;
case value2:
    statement2
    break;
...
case valueN:
    statementn
    break;
default: statement
}
    
```

Counters

```

purpose: to allow a variable to have its value changed
         based on it original value
syntax:  variable = variable operator expression;
example: int a = 10;
         a = a - 3;
    
```

While Loop

```

syntax: while (boolean-expression)
        statement
examples: int i=10;
         while (i > 0)
         {
             cout << i;
             i--;
         }

         (assume inFile is an ifstream variable)
         string s;
         getline(inFile, s);
         while (inFile)
         {
             cout << s;
             getline(inFile, s);
         }
    
```

Note: while(inFile) can be written while(!inFile.eof())

Do-While Loop

```

syntax: do
        statement
        while(boolean-expression);
example: int choice;
         do
         {
             cout << "Enter your choice: ";
             cin >> choice;
         }
         while (choice < 1 && choice > 5);
    
```

For-Loop

```

syntax: for(initial-statement; loop-condition; update-statement)
        statement
order of execution:  initial-statement
                    loop-condition
                    statement
                    update-statement
                    return to loop condition
example: for (int x=0; x<5; x++)
         {
             cout << x * x << ' ';
         }
    
```

Break Statement

```

purpose: exits the surrounding control structure
syntax:  break;
    
```

Continue Statement

```

purpose: short circuits the execution of the statement of a
         loop; the loop continues
syntax:  continue;
    
```

Single Dimension Arrays

declaration syntax: type identifier[size];
 type identifier[] = { value-list};
 type identifier[size] = { value-list};
 element access syntax: identifier[index-expression]
 examples: int a[100];
 a[0] = 33;

 double d[] = { 1.1, 2.2, 3.3};
 cout << d[1];

Two Dimension Arrays

declaration syntax: type identifier[size][size];
 type identifier[][] = { value-lists};
 type identifier[size][size]
 = { value-lists};
 element access syntax: identifier[index-exp][index-exp]
 examples: int a[100][200];
 a[0][15] = 33;

 double d[][] = { {1.1, 2.2, 3.3},
 {4.4, 5.5, 6.6}
 };
 cout << d[1][2];

Void Functions

prototype syntax: void function-identifier(parameter-list);
 function syntax: void function-identifier(parameter-list)
 {
 statements
 }
 function call syntax: function-identifier(actual-parameters);

Non-Void Functions

prototype syntax: type function-identifier(parameter-list);
 function syntax: type function-identifier(parameter-list)
 {
 statements
 at-least-one-return-statement
 }
 function call syntax: function-identifier(actual-parameters);

Return Statement

purpose: exit a function; may set a value to
 return from the function
 syntax: return;
 return expression;

Formal Parameter

parameter listed in a function header

Actual parameter

parameter listed in a function call

Pass-by-Value Parameter

a copy of the actual parameter is created under the name of the formal parameter; the formal parameter becomes a separate variable and changes to it cannot change the actual parameter

formal parameter syntax: nothing additional
 actual parameter syntax: nothing additional

Pass-by-Reference Parameter

the address of the actual parameter is passed to the formal parameter; the formal parameter becomes a local name for the memory address of the actual parameter; changes to the formal parameter are changes to the actual parameter

special formal parameter syntax: type & identifier
 special actual parameter syntax: nothing additional

Note: Arrays are pass by reference.

Const Functions

purpose: function does not change passed parameters
 syntax: function header ends in *const*

Const Parameters

purpose: function does not change passed parameter
 syntax: *const* precedes parameter type in header

Const Return Type

purpose: prevents address of returned type from being
 used to access the data structure in the function
 or class
 syntax: *const* precedes return type of function header

Parameter List

purpose: list of variables to be sent to a function
 in a call or to receive sent values or
 addresses in a function header
 actual parameter syntax: expression[, expression]
 formal parameter syntax: type [&] identifier[, type [&] identifier]

Array as actual Parameter

actual parameter: array-identifier

Array as Formal Parameter

when an array is a formal parameter, the left-most index may remain blank; arrays are always pass-by-reference, the ampersand (&) is not required

examples: int a[]
 double f[][MAX_COL]

Void Function Example

prototype: void ExampleFn(int, float &, float []);
 function: void ExampleFn(int size, float & avg, float c[])
 {
 float sum = 0;
 for (int x=0; x<size; x++)
 sum = sum + c[x];
 avg = sum / size;
 }
 call: float f[MAX], average;
 int size = 10;
 ...
 ExampleFn(size, average, f);
 cout << "Average: " << average;

Non-Void Function Example

prototype: float ExampleFn(int, float []);
 function: void ExampleFn(int size, float c[])
 {
 float sum = 0;
 for (int x=0; x<size; x++)
 sum = sum + c[x];
 return sum / size;
 }
 call: float f[MAX], average;
 int size = 10;
 ...
 average = ExampleFn(size, f);
 cout << "Average: " << average;

Access Modifiers

public: members of a collection that are public may be accessed from outside of the collection object; public members may be inherited
 private: members of a collection that are private may not be accessed from outside of the collection object; private members may be accessed only from inside of the collection; private members may not be inherited
 protected: members of a collection that are protected may not be accessed from outside of the collection object; private members may be accessed only from inside of the collection; protected members may be inherited

Struct

a collection of members, default access is public
 syntax: struct struct-name
 {
 type identifier;
 type identifier;
 ...
 type identifier;
 };
 declaration syntax: struct-name identifier;
 member access: identifier.member
 examples: struct Example
 {
 int i;
 double d;
 };
 Example a;
 a.i = 42;
 a.d = 5.3;
 Example b[MAX];
 b[0].i = 22;
 b[0].d = -3.1;

Classes

a collection of members, default access is private; members consist of member function methods and data member; member functions are implemented separately from the definition to facilitate information hiding;

definition syntax: class class-name
 {
 public:
 public-members
 protected:
 protected-members
 private:
 private-members
 };
 function method implementation syntax:
 type class-name::method-name(parameter-list)
 {
 statements
 }

Constructors

members with the same name as the collection; called when an object of the class type is created; constructors are always public

definition syntax: class-name(parameter-list);

implementation syntax:

```
class-name::class-name(parameter-list)
{
    statements
}
```

call syntax:

```
class-name variable-name;
class-name variable-name(parameter-list);
variable-name = class-name(parameter-list);
```

Destructors

members with the same name as the collection preceded by a tilde (~); called when an object of the class type is deleted or passes out of scope; destructors are always public

definition syntax: ~class-name();

implementation syntax:

```
class-name::~~class-name()
{
    statements
}
```

Class Example

definition example (in file example.h)

```
#ifndef _EXAMPLE
#define _EXAMPLE
using namespace std;
```

```
class Example
{
public:
    Example();
    Example(int init);
    int GetI();
private:
    int i;
};
```

implementation example (in file example.cpp)

```
#include "example.h"
using namespace std;
```

```
Example::Example()
{
    i = 0;
}
```

```
Example::Example(int init)
{
    i = init;
}
```

```
int Example::GetI()
{
    return i;
}
```

usage example (in file test.cpp)

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

```
int main()
{
    Example a;
    Example b(33);
```

```
a = Example(2);
```

```
cout << a.GetI() << endl; // outputs 2
cout << b.GetI() << endl; // outputs 33
```

```
return 0;
}
```

Inheritance

a collection may inherit the public and protected members of another class; the inherited members become members of the inheriting class; members inherited from the parent class may be redefined (overridden) in the child class

class inheritance syntax:

```
class child-class-name : access-modifier parent-class-name
{
    ...
};
```

Note: the access-modifier may determines how parent class members are inherited; public inherits public as public, protected as protected; protected inherits public and protected as protected; private inherits public and protected as private

constructor header syntax:

```
child-class-name(p-list) : parent-class-name(p-list)
```

Note: the members of the p-list of the parent class constructor must be members of the p-list of the child class constructor

constructor usage syntax:

```
child-class-name identifier(p-list);
```

Typical Creation of Random Numbers

```
#include <cstdlib>
#include <ctime>
```

```
srand((unsigned)time(NULL)); // seed generator
```

```
long num = rand(); // create random integer
```

Pointers

a variables whose content is an address

declaration syntax: type * identifier;

example: int *p;

NULL

pointer value that does not point to any memory address; any pointer can be assigned the constant NULL; any pointer can be tested against the constant NULL

Address of Operator

returns the address of a its operand

syntax: & operand

example: int i = 3;
 int *p = &i;

Dereferencing Operator

refers to the object indicated by the address in a pointer

syntax: * pointer-variable

example: int i = 3;
 int *p = &i;
 cout << *p; // outputs 3

Member Access Operator Arrow

refers to the member of an object indicated by the address in a pointer

syntax: collection-variable -> member

example: struct Ex
 {
 int a, b;
 };

 Ex m;
 Ex *p;

 p = &m;
 p->a = 123;

Dynamic Variables

dynamic variables are created a run time with the operator new; the memory used to create a dynamic variable must be returned when no longer needed with the operator delete; dynamic variables have no identifiers so they have no scope; only pointers may reference dynamic variables

operator new syntax: new type
 new type(parameter-list);
 new type[size]
 new type[size][size] ...;

operator new examples: int *p = new int;
 struct Ex
 {
 int a, b;
 };
 Ex *q;
 q = new Ex;
 q->a = 33;

operator delete syntax: delete pointer;
 delete [] pointer-to-array;

operators new and delete examples:
 int *p = new int;
 *p = 45;
 cout << *p;

```
struct Ex
{
    int a, b;
};
```

```
Ex *q;
q = new Ex;
q->a = 33;
```

```
Ex *r;
r = new Ex[10];
```

```
r[5].a = 99;
```

```
delete p;
delete q;
delete [ ] r;
```

Relational Operator Overloading for Classes

declare public; return boolean value

header syntax: bool operatorop(const type & a) const
example prototype: bool operator==(const Bob & a) const;
example method implementation:

```
bool Bob::operator==(const Bob & a) const
{
    if (name == a.name)
        return true;
    return false;
}
```

Assignment Operator Overloading for Classes

*declare public; return *this*

header syntax: const type & operator=(const type & a)
example prototype: const Bob & operator=(const Bob & a);
example method implementation:

```
const Bob & Bob::operator=(const Bob & a)
{
    name = a.name;
    height = a.height;
    weight = a.weight;
    return *this;
}
```

Output Stream Operator Overloading for Classes

declare friend in prototype only; no access modifier; return ostream identifier

prototype syntax:

```
friend ostream & operator<<(ostream & out, type & a);
```

header syntax:

```
ostream & operator<<(ostream & out, type & a) { ... }
```

example method implementation:

```
ostream & operator<<(ostream & out, Bob & a)
{
    out << a.name << endl
      << a.height << endl
      << a.weight << endl;
    return out;
}
```

Input Stream Operator Overloading for Classes

declare friend in prototype only; no access modifier; return istream identifier

prototype syntax:

```
friend istream & operator>>(istream & in, type & a);
```

header syntax:

```
istream & operator>>(istream & in, type & a) { ... }
```

example method implementation:

```
istream & operator>>(istream & in, Bob & a)
{
    string dummy;
    getline(in, a.name);
    in >> a.height >> a.weight;
    getline(in, dummy);
    return in;
}
```

Function Templates

syntax: template<class T>
 function definition

call syntax: function-name(parameter-list)
 function-name<type>(parameter-list)

Class Templates

syntax: template<class T>
 class definition

object declaration syntax:
 class-name<type> identifier;
 class-name<type> identifier(parameter-list);

Scope

Scope is the region where an identifier is said to be visible. Another way to think of this is that scope is the property that determines if a variable is usable or accessible in a specific context. All identifiers are said to have scope and only identifiers have scope. The scope of an identifier is said to be local, global or out of scope.

Identifiers that are local or global may be accessed. Identifiers that

are out of scope can not be accessed.

An identifier declared within a compound statement is local to that compound statement. This means that it cannot be accessed outside of that compound statement, as it would be out of scope. However, the same variable may be global to another compound statement if that second compound statement is within the compound statement in which the identifier was declared. An identifier that is global can be accessed.

Identifiers declared in the header of a function are considered local to that function. Identifiers declared in a For statement are considered local to the body of the For Loop.

Identifiers that are declared outside of a function are considered global within the file that they are declared in.

Access modifiers (public, protected and private) can be used to modify the access rights of local identifiers inside the compound statement of a class or struct. By default, all identifiers in a class are private and cannot be accessed from outside of the class. By default, all identifiers in a struct are public and can be accessed from outside of the class. Protected identifiers also cannot be accessed from outside of a class or struct. Since class and struct are types, an object must be created in order for any of their members to exist and (.) or (->) or () operators must be used to access public members of an object of a struct or class type.*

In addition, identifiers as said to have the property namespace. In CMPS 150 and 260, all variables as created and used within namespace std and all files should have the line "using namespace std;" following library includes.

The scope resolution operator (::) may be used to specify the class, struct or namespace of an identifier when necessary to avoid confusion or because overriding (local reuse of a global identifier) has resulted in an identifier being hidden from view.

Setting Namespace to std

```
using namespace std;
```

Typical Pre-compiler Instructions

```
#include <library>
#ifndef identifier
#define identifier
#endif
```

Common Libraries

iostream	- standard input and output; contains objects cin and cout
fstream	- file input and output; contains classes ifstream and ofstream
cstdlib	- standard library; contains srand, rand functions and many other functions
string	- class string
iomanip	- output manipulation functions setw and setprecision
ctime	- time functions; contains function time
cmath	- math functions