C++ Reference Card

C++ Data Types

Data Type	Description
`bool`	boolean (true or false)
`char`	character ('a', 'b', etc.)
`char[]`	character array (C-style string if `null` terminated)
`string`	C++ string (from the STL)
`int`	integer (1, 2, -1, 1000, etc.)
`long int`	long integer
`float`	single precision floating point
`double`	double precision floating point

These are the most commonly used types; this is not a complete list.

Operators

The most commonly used operators in order of precedence:

1	++ (post-increment), -- (post-decrement)
2	! (not), ++ (pre-increment), -- (pre-decrement)
3	*, /, % (modulus)
4	+, -
5	<, <=, >, >=
6	== (equal-to), != (not-equal-to)
7	&& (and)
8	|| (or)
9	= (assignment), *=, /=, %=, +=, -=

Console Input/Output

`cout <<`	console out, printing to screen
`cin >>`	console in, reading from keyboard
`cerr <<`	console error

Example:

```
cout << "Enter an integer: ";
cin >> i;
cout << "Input: " << i << endl;
```

File Input/Output

Example (input):

```
ifstream inputFile;
inputFile.open("data.txt");
inputFile >> inputVariable;
// you can also use get (char) or
// getline (entire line) in addition to >>
  ...
inputFile.close();
```

Example (output):

```
ofstream outFile;
outfile.open("output.txt");
outFile << outputVariable;
  ...
outFile.close();
```

Decision Statements

if

```
if (expression)
    statement;
```

Example

```
if (x < y)
    cout << x;
```

if / else

```
if (expression)
    statement;
else
    statement;
```

Example

```
if (x < y)
    cout << x;
else
    cout << y;
```

switch / case

```
switch(int expression)
{
  case int-constant:
    statement(s);
    break;
  case int-constant:
    statement(s);
    break;
  default:
    statement;
}
```

Example

```
switch(choice)
{
  case 0:
    cout << "Zero";
    break;
  case 1:
    cout << "One";
    break;
  default:
    cout << "What?";
}
```

Looping

while Loop

```
while (expression)
    statement;

while (expression)
{
    statement;
    statement;
}
```

Example

```
while (x < 100)
    cout << x++ << endl;

while (x < 100)
{
    cout << x << endl;
    x++;
}
```

do-while Loop

```
do
    statement;
while (expression);

do
{
    statement;
    statement;
}
while (expression);
```

Example

```
do
    cout << x++ << endl;
while (x < 100);

do
{
    cout << x << endl;
    x++;
}
while (x < 100);
```

for Loop

```
for (initialization; test; update)
    statement;

for (initialization; test; update)
{
    statement;
    statement;
}
```

Example

```
for (count = 0; count < 10; count++)
{
    cout << "count equals: ";
    cout << count << endl;
}
```

Functions

Functions return at most one value. A function that does not return a value has a return type of `void`. Values needed by a function are called parameters.

```
return_type function(type p1, type p2, ...)
{
    statement;
    statement;
    ...
}
```

Examples

```
int timesTwo(int v)
{
    int d;
    d = v * 2;
    return d;
}

void printCourseNumber()
{
    cout << "CSE1284" << endl;
    return;
}
```

Passing Parameters by Value

```
return_type function(type p1)
```

Variable is passed into the function but changes to *p1* are not passed back.

Passing Parameters by Reference

```
return_type function(type &p1)
```

Variable is passed into the function and changes to *p1* are passed back.

Default Parameter Values

```
return_type function(type p1=val)
```

val is used as the value of *p1* if the function is called without a parameter.

Pointers

A pointer variable (or just pointer) is a variable that stores a memory address. Pointers allow the indirect manipulation of data stored in memory.

Pointers are declared using *. To set a pointer's value to the address of another variable, use the & operator.

Example

```
char c = 'a';
char* cPtr;
cPtr = &c;
```

Use the indirection operator (*) to access or change the value that the pointer references.

Example

```
// continued from example above
*cPtr = 'b';
cout << *cPtr << endl;  // prints the char b
cout << c << endl;      // prints the char b
```

Array names can be used as constant pointers, and pointers can be used as array names.

Example

```
int numbers[]={10, 20, 30, 40, 50};
int* numPtr = numbers;
cout << numbers[0] << endl;     // prints 10
cout << *numPtr << endl;        // prints 10
cout << numbers[1] << endl;     // prints 20
cout << *(numPtr + 1) << endl;  // prints 20
cout << numPtr[2] << endl;      // prints 30
```

Dynamic Memory

Allocate Memory

```
ptr = new type;

ptr = new type[size];
```

Examples

```
int* iPtr;
iPtr = new int;

int* intArray;
intArray = new int[5];
```

Deallocate Memory

```
delete ptr;
delete [] ptr;
```

Examples

```
delete iPtr;
delete [] intArray;
```

Once a pointer is used to allocate the memory for an array, array notation can be used to access the array locations.

Example

```
int* intArray;
intArray = new int[5];
intArray[0] = 23;
intArray[1] = 32;
```

Structures

Declaration

```
struct name
{
  type1 element1;
  type2 element2;
};
```

Example

```
struct Hamburger
{
  int patties;
  bool cheese;
};
```

Definition

```
name varName;

name* ptrName;
```

Example

```
Hamburger h;

Hamburger* hPtr;
hPtr = &h;
```

Accessing Members

```
varName.element=val;

ptrName->element=val;
```

Example

```
h.patties = 2;
h.cheese = true;

hPtr->patties = 1;
hPtr->cheese = false;
```

Structures can be used just like the built-in data types in arrays.

Classes

Declaration
```
class classname
{
public:
  classname(params);
  ~classname();
  type member1;
  type member2;
protected:
  type member3;
private:
  type member4;
};
```

Example
```
class Square
{
public:
  Square();
  Square(float w);
  void setWidth(float w);
  float getArea();
private:
  float width;
};
```

public members are accessible from anywhere the class is visible.

private members are only accessible from the same class or a friend (function or class).

protected members are accessible from the same class, derived classes, or a friend (function or class).

constructors may be overloaded just like any other function. You can define two or more constructors as long as each constructor has a different parameter list.

Definition of Member Functions
```
return_type classname::functionName(params)
{
    statements;
}
```

Examples
```
Square::Square()
{
    width = 0;
}

void Square::setWidth(float w)
{
    if (w >= 0)
      width = w;
    else
      exit(-1);
}

float Square::getArea()
{
    return width*width;
}
```

Definition of Instances
```
classname varName;

classname* ptrName;
```

Example
```
Square s1();
Square s2(3.5);

Square* sPtr;
sPtr=new Square(1.8);
```

Accessing Members
```
varName.member=val;
varName.member();

ptrName->member=val;
ptrName->member();
```

Example
```
s1.setWidth(1.5);
cout << s.getArea();

cout<<sPtr->getArea();
```

Inheritance

Inheritance allows a new class to be based on an existing class. The new class inherits all the member variables and functions (except the constructors and destructor) of the class it is based on.

Example
```
class Student
{
public:
  Student(string n, string id);
  void print();
protected:
  string name;
  string netID;
};

class GradStudent : public Student
{
public:
  GradStudent(string n, string id,
                string prev);
  void print();
protected:
  string prevDegree;
};
```

Visibility of Members after Inheritance

Inheritance Specification	Access Specifier in Base Class		
	private	protected	public
private	-	*private*	*private*
protected	-	*protected*	*protected*
public	-	*protected*	*public*

Operator Overloading

C++ allows you to define how standard operators (+, -, *, etc.) work with classes that you write. For example, to use the operator + with your class, you would write a function named `operator+` for your class.

Example
Prototype for a function that overloads + for the `Square` class:
```
Square operator+ (const Square &);
```

If the object that receives the function call is not an instance of a class that you wrote, write the function as a friend of your class. This is standard practice for overloading << and >>.

Example
Prototype for a function that overloads << for the `Square` class:
```
friend ostream & operator<<
          (ostream &, const Square &);
```

Make sure the return type of the overloaded function matches what C++ programmers expect. The return type of relational operators (<, >, ==, etc.) should be `bool`, the return type of << should be `ostream &`, etc.

Exceptions

Example
```
try
{
  // code here calls functions that might
  // throw exceptions
  quotient = divide(num1, num2);

  // or this code might test and throw
  // exceptions directly
  if (num3 < 0)
    throw -1;  // exception to be thrown can
               // be a value or an object
}
catch (int)
{
  cout << "num3 can not be negative!";
  exit(-1);
}
catch (char* exceptionString)
{
  cout << exceptionString;
  exit(-2);
}
//  add more catch blocks as needed
```

Function Templates

Example
```
template <class T>
T getMax(T a, T b)
{
  if (a>b)
    return a;
  else
    return b;
}


// example calls to the function template
int a=9, b=2, c;
c = getMax(a, b);

float f=5.3, g=9.7, h;
h = getMax(f, g);
```

Class Templates

Example
```
template <class T>
class Point
{
public:
  Point(T x, T y);
  void print();
  double distance(Point<T> p);
private:
  T x;
  T y;
};


// examples using the class template
Point<int> p1(3, 2);
Point<float> p2(3.5, 2.5);
p1.print();
p2.print();
```

Suggested Websites

C++ Reference: http://www.cppreference.com/ http://www.informit.com/guides/guide.aspx?g=cplusplus

C++ Tutorial: http://www.cplusplus.com/doc/tutorial/ http://www.sparknotes.com/cs/

C++ Examples: http://www.fredosaurus.com/notes-cpp/

Gaddis Textbook:
- Video Notes http://media.pearsoncmg.com/aw/aw_gaddis_sowcso_6/videos
- Source Code ftp://ftp.aw.com/cseng/authors/gaddis/CCSO5 (5th edition)