**Declarations:** int e=0, f, g[120], h[200]
typedef *originalType newType*
#define *returnChar* \n

**Sizes:** Float≥32, Double≥48, long≥32, short=16

0x12345678 (memory, hex);
012345 (octal)

**I/O: std::cin**
**getline(name,20,delim):** read until 19 chars has read, a new line/EOF/delim is encountered. '\n' character is discarded
**get(name,20):** similar to getline, but keeps '\n'
**>>:** Stop at whitespace, remaining stays in buffer (including \n). If string empty, read another one
**get():** read single character

struct *structName* {
int a, int b} *variableName* **;**
union *unionName* {
int a, float b} *variableName***;**
enum *typeName* { a=1, b=10};
(casting required for int => enum only)
(*enum typeName(15);* is fine, in range)

**Pointers**
**declaration**(valid):
int a, *p=&i, b; means int* p=&i
**otherwise**(invalid): *p = &i; ≡ assigning the location as the value of p
**p+1:** means adding 4 to memory address if sizeof(int) = 4

**void* a=b** does not need casting
int* b =a needs casting

**Function Pointers:**
Declaration:
*return_type* (**functionPointerName*) (*arguments)*;

Casting:
*return_type* (*) (*arguments)*
*variableName*;

**Array**
int foo[][2]={{1,2}, {3,4}}
**Multidimensional**: arrays of arrays. (not pointers to pointers)
-Represented in memory as one-dimensional array

| [0][0] | [0][1] | [1][0] | [1][1] |
|---|---|---|---|

arrayname[i] ≡ *(array+i)

**USE *const* whenever appropriate**
const int* a = &j – cannot change value of i
int* const a = &j – cannot point to other things

-Assigning the address of *non-const* data to *pointer-to-const* is ok, unless data is a pointer

**Function**
-Return value and arguments are by value
-Temporary variables are created when the arguments do not match exactly, or when the arguments are expressions(non-Lvalues)
**-default values:** start from right:
void foo(int a, int b=666);

**Function Templates**
template <class Any>
void foo(Any a, Any b);

template <class Any>
void foo(Any a, Any b){ … }

**Implicit Instantiation:** foo(1,2);
**Explicit Instantiation:**
template void foo<int>(int, int);
**Specialization:**
-Old:
void foo<int>(int a, int b);
void foo<int>(int a, int b){ … }
=use regular prototype=
void foo(int a, int b){ … }

-Current:
template <> void foo<int> (int a, int b);
= or =
template <> void foo(int a, int b);

**Preprocessor**
#error "*error message*"
#line *lineNumberToBeAssumed*
#pragma GCC poison printf
__LINE__
__FILE__
__ cplusplus

**Object like definition:**
#define foo
#define PI 3.14
#define greet "hi"

**Function like definition:**
#define max( X, Y ) (( X >= Y ) ? ( X ) : ( Y ))

**Classes**
Example Declaration:
class Stock{
private:
static const int a = 10;
void foo() const;
}

void Stock::foo() const { … }

Implicit Member Functions
-Default Constructor
UserClass::UserClass(): mem(a), mem(b){ … }
**-Constant data members and references must be initialized this way**

-Copy Constructor
**-Called when:**
-new object instantiated to same class
-object is passed to function by value
-function returns object by value
-compiler generates a temp. object
**-Provide explicit copy constructor when *new* operator is used in the class**

-Assignment =
UserClass& UserClass::operator=(const UserClass& cn){
if (this==&cn) return *this;
delete c_pointer;
c_pointer = new type;
….
return *this;
}

-Default Destructor
-Address Operator

**Class Conversions**
class UserClass;
UserClass a = 23; is valid if
UserClass::UserClass(int a); is defined.
-This behavior can be avoided with *explicit*

UserClass a;
int i = a; is valid if
operator int() const; is defined.
-No return value in declaration, but must include *return* in definition

**Friends**
class Foo{ …
friend ostream& operator<<(ostream& os, const Foo& f);
}

*See advanced templates for more friends*

**Pure Virtual Functions**
virtual *FcnName* (int a ,… ) = 0;
-Derived class must provide definitions to pure virtual functions
-Classes with pure virtual functions become abstract base class, and no instance can be created

Created by Vincent Chu

http://www.sfu.ca/~vwchu
chuvincent (at) hotmail (dot) com
(must include "discretemath" as subject line in e-mail or it will be filtered)

**Memory Models**

| | Scope | |
|---|---|---|
| Static with internal linkage | File | Outside of all functions with keyword *static* |
| Static with external linkage | File | Outside of all functions (use *extern* for redeclaration) |
| Static with no linkage | Block | In block with keyword *static* |

**Memory Definitions**:
**Automatic**: Stack (in fuctions)
**Static**: throughout the execution of program
**Dynamic**: new/delete, in free store
**Volatile:** variable can be changed by other programs
**Mutable:** member variable that can be changed even though the class is constant

**Namespace**
**Unnamed namespace:** available until the end of declarative region(replaces static internal linkage)
**using namespace std;** Duplicate variables can be hidden by location def'n
**using std::cin;** Duplicate variables will generate a compiler error

**Operator Overloading**

```
Foo Foo::operator+(const Foo& f) const{
  … return sum; //a copy will be returned
}

Foo Foo::operator-() const {
  … unary negate
}

const char& String::operator[](int i)
const;
char & String::operator[](int i);
```

new/delete

```
void* operator new( size_t iValue ) ;

void operator delete( void* pRecord );

void* operator new[]( size_t iValue );
void operator delete[]( void* pRecord );
```

**Single Inheritance**

```
class DerivedClass: public ParentClass {
  …
};
```

-Derived class will not overload functions with the same name in the base class, instead it hides all the functions with the same name

Derived class contains Dynamic Memory Allocation (DMA)
**Copy Constructor:**

```
hasDMA::hasDMA(const hasDMA& hs):
  baseDMA(hs){ … };
```

**Operator =** :

```
hasDMA& hasDMA::operator=(const
hasDMA& hs){
  if (this==&hs) return *this;
  baseDMA::operator=(hs);
….
  return *this; }
```

**Multiple Inheritance**
-problems:
  -member function ambiguity
  -containment multiplication

Virtual Base Class

```
class Car: virtual public Vehicle {
  …
}
class Truck: virtual public Vehicle {
  …
}
class SUV: public car, public Truck {
  …
}
```

*SUV::SUV*(): *Truck*(), *Car*(), *Vehicle*(){…}
**-initialize the virtual base class, or default constructor will be used**

**Class Templates**
-must stay in one file, or use *export*
Declaration

```
template <class Type, int maxSize, class
Type 2 = int>
class Stack{ …Stack(); void foo();… };
```

Constructor Definition

```
template<class Type, int maxSize, …>
Stack<Type, maxSize>::Stack(){ … }
```

Member Function Definition

```
template<class Type, int maxSize, …>
void Stack<Type, maxSize >::foo(){ … }
```

Static data member initialization

```
template<class T> int Stack<T>::data=0;
```

**Using templates**
**Implicit Instantiations:**

```
Stack <int, 10> stuff;
```

-No code is generated until an object is required (pointer only doesn't count)

**Explicit Instantiation:**

```
class Stack<int,10>;
```

-no object is being created

**Explicit Specialization**:

```
template <> Stack<int, 2, int> { … };
```

**Partial Specialization:**

```
template <class T> class Stack<T, 2,
int){ … };
```

**Advanced Class Templates**

Nested Templates
-Template class contains a template class

Template as parameters

```
template <template <type T> class Thing> class
Crab {
        private:  Thing<int> s1;
                  Thing<double> s2;
        …
};
```

Bound Template Friend Functions
-each class specialization gets a matching specialization for a friend

```
template <typename T> void report(T&);

template <typename TT>
class HasFriendT {
  { …
friend void report <HasFriend <TT> >
(HasFriend<TT>& );
};
```

Unbounded Template Friend Functions
-every function specialization is a friend to every class specialization

```
template<typename T>
class ManyFriend{  …
        template<class C, class D> friend void
show2(C&, D&);
};
```

**Exceptions** #include <exception>
Function Declarations

```
void foo() throw();
//foo doesn't throw exceptions

void foo() throw(const char*, exception);
//foo throws const char*, exception inst.
```

Rethrowing the exception

```
void foo(){
  try {    …    }
  catch (char* char) { …
     throw();
       //rethrow the same exception  }
```

Catch anything: catch( …)

Sample Code:

```
class problem{ … };
…
void super() throw (problem&){
 … if (oh_no){
     { problem oops();
       throw oops(); } //a copy is thrown
       }
}
….
try { super(); }
catch (problem& p) { p.what()…. }
```

**RTTI**
dynamic_cast: return 0 if unsuccessful down cast
reinterpret_cast
static_cast: performed allowed conversions, including from base class to derived class
const_cast: get rid of *const*