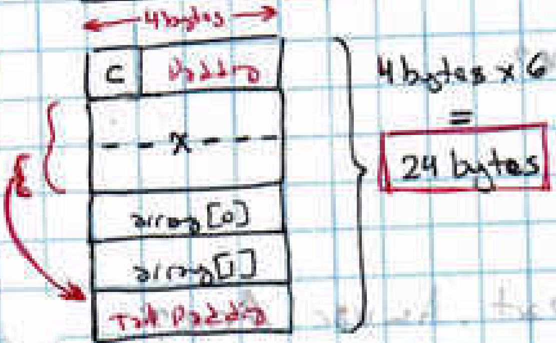
structs

①
```
struct A {
    char c;
    double x;
    int array [1];
};
```

A) What's the size of A?



A. Padding between member of a structure is called internal padding

B. char 1 byte
short 2 bytes
~~long int~~

int	4 bytes
long long	8 bytes
float	4 bytes
double	8 bytes
pointer	4 bytes
enum	4 bytes

C. Padding between the last element and the end of the space occupied by the struct ro is called tail padding
It may required if the last element does not end on the appropiate boundary size

B) Can you do:

a. A a1, a2;
b. a1 = a2 ← Assignment: yes (C/c++)
if (a1 = a2){ ← Equality: No you can only compare individual fields
... ej: a1.c == a2.c && a1.x == a2.x ...
}

C) Do they work with I/O operators? ej cout << a1 << endl;?
No. only with individual fields ej cout << a1.c << endl.

②

```
struct A {
  char A_var;
};
struct B {
  A a_part;
  int B_var;
};
struct C {
  B b_part;
  int C_var;
};
```

```
struct D {
  C c_part;
  char c;
  int i;
};
```

```
~~in the same~~
struct D
  int B_var;
  A a_part
};
```

A a
C c

A. (char *) &a == &a.A_var //ok?

yes.

~~B. (char *) &c == &c_part~~

B. (char *) &c == &c.b_part.a_part.A_var //ok?
yes.

C. (int *) &c == &c.b_part.B_var. //ok?
No.

~~D. (int *) &c == &c.C_var //ok?~~

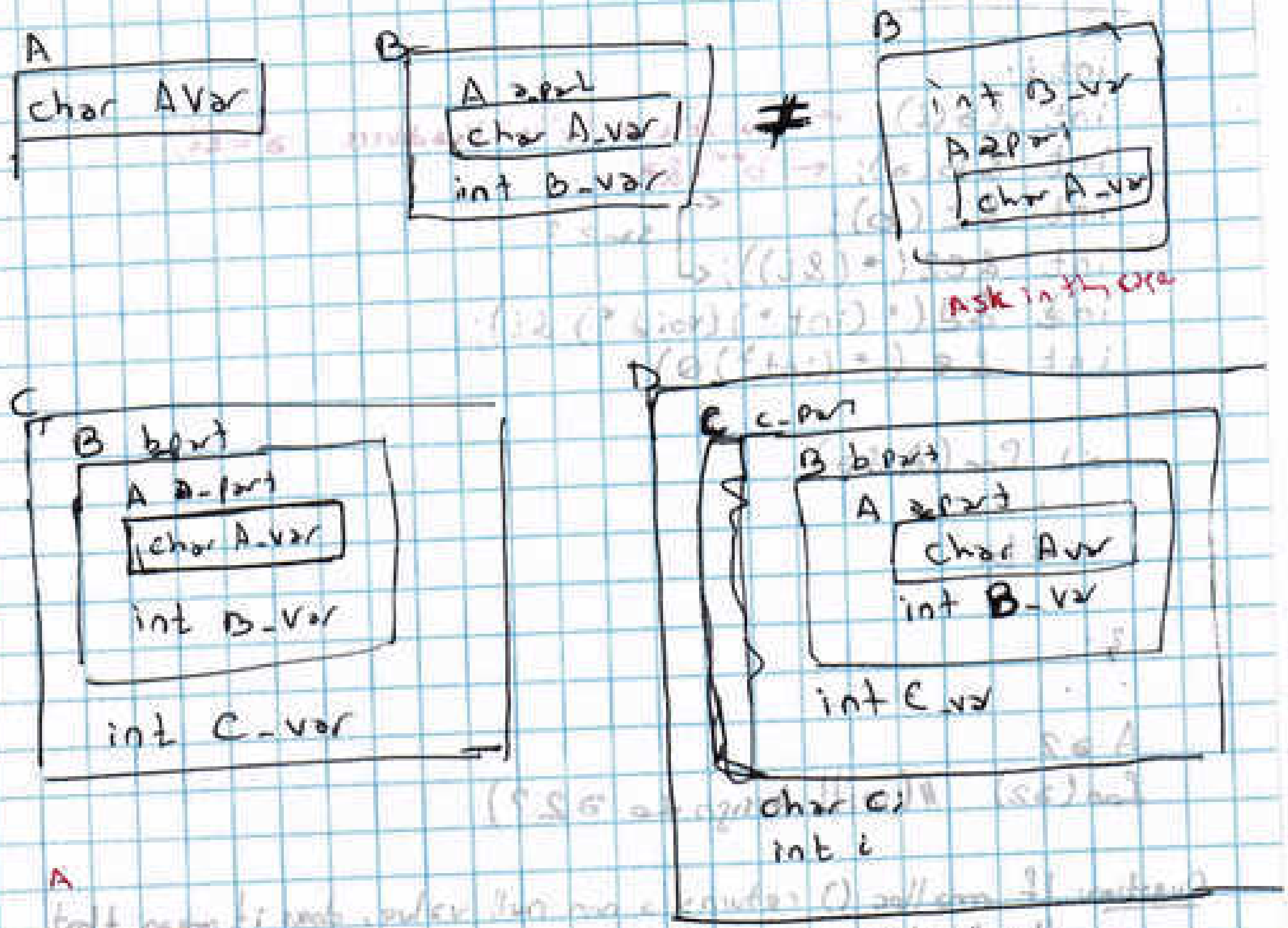
D. D d;
(char *)&d == &d.c &&
(int *) &d == &d.i //ok?

No

E. (char *) &d == &d.c ? No

F. (char *) &d == &d.cpart.b_part.a_part.A_var

~~G. ((char~~



L-values and R-values

1. expression are either lvalues or Rvalues
2. An lvalue refers to an object or function
   eg. all variables (include non modifiable (const)) are lvalues
3. An rvalue is a temporary value that does not persist beyond the expression that uses it.

$$\underbrace{\text{int } x}_{\text{lvalue}} = \underbrace{3+5}_{\text{rvalue}}$$

References in C++

```
int i;
int &a(i);  ← variable a holds i's address.  a* = &i;
int &b(a);  ← b** = &a;
int &c(b);        ] same ?
int &c2(*(&i));
int &d(*(int*)(void*)&i);
int &e(*(int*)0);

void foo(A &a)
 ...
 a = b;
};
...
A a2
foo(a2)  //(will Assign to a2?)
```

Question: If malloc() returns a non null value, does it mean that the allocation succeded?

~~yes. However, it could~~

yes. The non-null value is a pointer to the allocated memory.
if malloc() return NULL could be:
- 1. An error
- or
- 2. Succeded to call malloc but with size zero.

Q What will this print out?

```
#include <iostream>
int main(){
    char i = 0;
    i--;
    std::cout << (int) i << std::endl;
}
```

R: It print out -1

Q: Does the compiler vendor need to tell you if char is signed or unsigned?
R: yes, it is known as implementation-defined behavior

Q: What will this print out?

```
void foo (int i1, int i2) {}
...
foo (printf ("First.\n"), printf ("Second.\n"));
```

R: we don't know. This could be an unspecified behavior
However, when compiled stdprint out: Second
First

Q: what will this do?

```
int *a = 0;
*a = 1;
```

R: Anything. This is known as unspecified behavior

Assertions Vs. Exceptions Vs Special Return Values

- Error code vs exceptions:
  a. Exception are more robust
  b. Resultcode can be ignored
  c. return true or false only good to report good or bad
- Assert/log are good for debugging techniques but not good idea for reporting mechanism to users/clients.

- Global error condition flag such used in errno() and perror() good but no for client/users
  a. However: error checks bulks up code, many less efficient & harder to read

- Non-local gotos such as setjump() and longjmp()
  a. setjump(): saves a known good state in a program
  b. longjmp(): Restore that state
  c. Problem with them: a. High coupling between setjmp & longjmp locales
     b. C++: they do not call destructors so there is no object cleanup. Therefore error recovery is almost impossible.

- Exceptions:
  a. A caller catches the exception
  b. Exception unwind to the stack frame of a catcher
  c. An exception is aware of C++ object & their destructors
  d. Exceptions are objects: eg: Range() invokes a constructor to create a Range object

e. Exception Example:

```
void foo (Vector &v){
   int i;
   ...                ← try block
   try { bar(v) } catch (Vector::Range){
     ... if try causes an exception then catch a Range object & do something
   }

void bar (Vector &v) { v[v.size()+1]; // trigger a range error);
```

Why is exception handling superior to traditional error handling techniques?

a. Instead to terminating the program, we can write a more robust, fault tolerant code

b. Instead of returning a value representing error, we can make a more readable

c. Instead of returning a legal value & leaving the program in an illegal state (which can keep running & cause mysterious crashes later on)

d. Developers are forced to program to run exceptions

e. run destructors

~~Reto~~

Run Time type Information (RTTI)

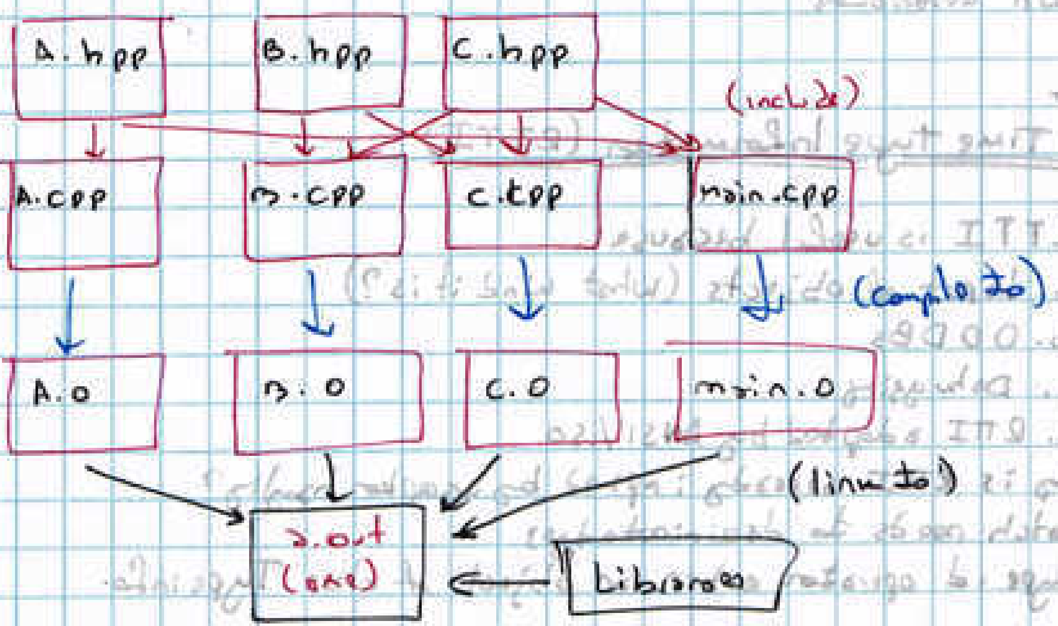
1. RTTI is useful because:
   a. Input of objects (what kind it is?)
   b. OODBs
   c. Debugging
   d. RTTI adopted by ANSI/ISO
2. Why is RTTI already implied by exception handling?
   catch needs to discriminate types
3. type id operator return an object of class Typeinfo.

4. Code fragment demonstrating RTTI:

```
#include <iostream.h>
#include <typeinfo.h> //Class Typeinfo
class A{};
void main{
  char ch; float x;
  if (typeid (ch) != typeid (x) // compare typeid @ run time
     cout << "ch & x are not the same type" << endl;
  cout << typeid (ch).name ()<<endl; //output char
  cout <<typeid (x). name() <<endl //output float
  cout << typeid (A). name() <<endl //output "A"
```

5. Stroustrup
   a. Resisted RTTI, arguing that it would lead to poor programs
   b. Bjarne Stroustrup is the designer & original implementer of C++

6. Why might RTTI undermine the use of virtual functions?
   a. temptation: lots of if-then-else or switch statements testing typeid
   b. OO solution: let dynamic binding figure out the type for you



Translation unit

Translation unit is the result of reading in a file, after all processing of include files and conditional compilation.

a.
```
void foo(){
  goo();
}
...
```

b. unclear: "the call to goo() will be syntax error if there is no declaration of it in this file"

✗ Unambiguous: "The call to goo() will be syntax error if this file doesn't declare goo(), and this file doesn't (recursively) include any header files that declare goo()."

✓ Translation unit: "This will be syntax error if there is no declaration of goo() in the translation unit."

type casting

1) Type-casting is called when converting an expression of a given type into another type

2) Implicit conversion do not required any operator.
a) they are performed automatically by copying the value into a compatible one.
ej: short a = 2000; } standard conversion affect fundamental data type
int b = a; } such as numerical types (short to int, int to float)
b) short to int, int to float, double to int
c) These conversions may imply a loss of precision
d) loss of precision may make the compiler to signal a warning
e) Avoid implicit conversion w/ Explicit convertions

3) Explicit convertions
a. there are two type of convertions: functional and c-like casts
b. Allow to convert any pointer into any other pointer types they point to. However, used indiscriminately can produce a code syntatically correct but producing runtime errors:
example:

```
#include <iostream>
using namespace std;
class CDummy{
  float i,j;
}
class CAddition{
   int x,y;
   public:
     CAddition (int a, int b){ x=a, y=b;}
     int Result (){ return x+y;}
};
int main(){
   CDummy d;
   CAddition *padd;
   padd = (CAddition *) &d;
   cout << padd -> result ();
   return 0;
}
```

② result will produce either run-time error or an unexpected result

① The program declares a pointer CAddition but then it assigns to it a reference to an object of another incompatible type using explicit type-casting.

c. traditional typecasting equivalents are:
i) (new_type) expression
ii) new_type (expression)

4) <u>Specific Casting operators</u>

expression to be converted

a. xxxx_cast <new_type> (expression)

b. <u>dynamic_cast <new_type> (expression)</u>

i) Only with pointers and references to objects
ii) Ensure that the result of the type conversion is a valid complete object of the requested class
iii) Always successfull when casting a class to one of its base classes
iv) <u>Example:</u>

```
class CBase {};
class CDerived : public CBase(){};
CBase base; CBase *p_base;
CDerived derived; CDerived *p_derived;
```

[OK] → p_base = dynamic_cast <CBase *> (& derived); ✓
↳ Derived → base ✓

[wrong] → p_derived = dynamic_cast <CDerived *> (& base); ✗
↳ base → derived ✗ produce compilation error.

v) Base-to-derived are not allowed w/ dynamic_cast unless the base class is polymorphic.
<u>example of polymorph class:</u>

```
#include <iostream>
#include <exception>
using namespace std;
class CBase {virtual void dummy() {}};
class CDerived: public CBase {int a;};
int main(){
   CBase   *p_baseA = new CDerived;
   CBase   *p_baseB = new CBase;
   CDerived   *p_derived;
 if ((p_derived = dynamic_cast <CDerived *> (p_baseA))) == 0)
   cout << "Null pointer on first type-cast" << endl;
 if ((p_derived = dynamic_cast <CDerived *) (p_baseB)) == 0)
   (cout << "Null pointer on second cast" << endl
```
[print out]

The code tries to perform both dynamic_cast from pointer object type CBase* (p_baseA and p_baseB) to pointer object of type CDerived*, but only the first one is successful

while p_baseA points to a full object of class CDerived, p_baseB is pointing to an object of class CBase which is an incomplete object of class CDerived

vi) Dynamic_cast cannot cast a pointer to an object that is not complete.
vii) If dynamic_cast is used to convert to a reference type and the conversion is not possible, an exception of type bad_cast is thrown instead.

vii) Dynamic_cast can also cast null pointers even between pointers unrelated classes, and also can cast pointers of any type to void pointers (void*)
viii) dynamic cast works on any class w/ at least one virtual function

c) static_cast <new type> (expression)

i) conversions between pointers to related classes.
ii) conversion from base-to-derived and derived-to-base
iii) Ensures that at least the classes are compatible if the proper object is converted
iv) However, no check is performed at run-time to check if the object being converted is in fact a full object of the destination type. It is up to the programmer to ensure that the conversion is safe.
v) the overhead of the type-safety of dynamic cast is avoided.
vi) example code:

```
class CBase{};
class CDerived : public CBase{};
CBase *p_base = new CBase;
CDerived *p_derived = static_cast<CDerived*>(p_base);
```

This is valid, although p_derived points to an incomplete object of the class and would lead to runtime errors if dereferenced.

d) reinterpret_cast <newtype> (expression)

i) Converts any pointer type to any other type, even of unrelated classes.
ii) simple binary copy of the value from one pointer to the other.
iii) All pointer conversion are allowed: neither the content pointed nor the pointer type itself is checked
iv)

e) Const_cast <newtype> (expression)

i) Manipulates the constness of an object, either to be set or to be removed.
ii) Example: pass a const argument to a function that expect a non-constant parameter

```
#include<iostream>
using namespace std;
void print (char *str){
    cout << str << endl;
}
int main(){
    const char *c = "sample text";
    print (const_cast <char *> (c));
    return 0;
}
```

f) <u>type_id</u>

i) return a reference to a constant object type type_info
ii) the returned value can be compared with another with operator == and !=
or can serve to obtain a null-terminated character sequence
representing the data type or class name by using its name()
member.
iii) when used w/ classes, RTTI can use it to keep track of dynamic objects
iv) when type-id is applied to an expression whose type is polymorphic class,
the result is the type of the most derived complete object.
v) <u>example code:</u>

```
#include <iostream>
#include <typeinfo>
using namespace std;
class CBase {virtual void f() {}};
class CDerived : public CBase {};

int main(){
    CBase *p_baseA = new CBase;
    CBase *p_baseB = new CDerived;
    cout << "a is:" << typeid(p_baseA).name() << endl;
    cout << "b is:" << typeid(p_baseB).name() << endl;
    cout << "*a is:" << typeid(*p_baseA).name() << endl;
    cout << "*b is:" << typeid(*p_baseB).name() << endl;
}
```

output

```
a is: class CBase *
b is: class CBase *
*a is class CBase
*b is class CDerived
```

I) The type that type-id considers for pointers is the pointer type itself
(a & b are CBase*)
II) However, when typeid is applied to objects (*a, *b) typeid yields
their dynamic type
vi) if typeid evaluates a pointer preceded by the dereference operator (*), and
this pointer has a null value, typeid throws a bad_typeid exception

One Definition Rule (ODR)

1) Each variable can be defined only once
2) You can declare a global variable by using extern
3) Defining creates a variable
4) Declaring (using extern) state the existence of a variable (it was declared somewhere else)

Struct vs classes / Class definition

```
class type1 { int mem 1 ;} o1;
class type2 { int mem 1 ;} o2;

o1 = o2   // works? No. types in C++ are by name
```

```
typedef Type1 type2
type1 o1;
type2 o2;

o1 = o2 // works? yes typedefs are aliases
```

```
class A {
    inline void f();
}
inline void A::f() {...}  // ← Does this go in header file or cpp file?
```

Friends.

```
class A {
    friend void foo();
  private:
    int i;
};
void foo(){
    A a;
    a.i; //OK!
}
```

```
void goo(){
    A a;
    a.i // wrong access violation
```

```
class A{
  friend int B::f();
  friend class B;
};
```

Public vs public:

design level ↑ ≠ ↑ implementation level

Forward Declaration

Which one is allowed?

```
(A) class A{
      int i;
      A * a;
    }
```

```
(C) class A{
      int i;
      static A a_number;
    }
```

```
(E) class A{          class B{
      int i;            int i;
      B b;              A a;
    }                 }
```

```
(B) class A{
      int i;
      A a;
    }
```

```
~~(D) class A{~~
      int i;
      B b;
    };
```

```
class A{
  int i;
  B *bptr;
}

class B{
  int i;
  A* aptr;
};
```

using forward declaration

```
Class B;
Class A {
  B * bptr;
}
Class B {
  A * aptr;
}
```

Classes define slope

scope is about what a name refers to

→ Resolution operators : Allow to refere a member

```
class A { ... };
int i = A ::var;  ← would this compile?
```

Scope Resolution Operator ::

1. used to qualify hidden names so you can still use them

2. The unary scope operator can be use if a namespace scope or global scope name is hidden by an explicit declaration of the same name in a block or class. eg:

```
int count = 0;
int main(void){
    int count = 0;
    :: count = 1;   //set global count to 1.
    count = 2;      //set Local count to 2.
    return 0;
}
```

Static Class Members

```
class A { int id;}
int obj_id ; // Global counter
A ::A(){
   id = obj_id++;
}
```

globals should be avoided

Better:

```
Class A {
  private:
     const int _id;
     static  int next_id;
};

// In cpp
A::A():id(next_id++){ }
int A:: next_id;
```

Encapsulation: Hiding stuff that is private to the implementation
User can't mess w/ implementation
User doesn't depend on implementation

Abstraction: set of interfaces

Public: All have access

Private: Only members functions of the same class or member functions of friend class

Protected: only member functions of the same class, friend classes or derived classes have access

---

```
class A { ... };
```

3. The class scope operator can be used to qualify class names or class member names.

```
#include <iostream>
using namespace std;
class X {
   public:
      static int count;
};
int X :: count = 10;  // define static data member

int main (void){
   int X = 0;     // hides class type X
   cout << "main X: " << X << " - X class count:" << X::count << endl;
}
```

Copy constructors, ~~Assignment operators~~

1. special Constructor for a class / struct used to make a copy of an existing instance.

2. example of copy constructor instances:

    a. MyClass (const MyClass & myClassOther);

    b. MyClass (MyClass &otherClass);

    c. MyClass (volatile const MyClass &otherClass);

    d. MyClass (volatile MyClass &otherClass);

3. The following are NOT copy constructors

    a. MyClass (MyClass *otherClass);

    b. MyClass (~~My~~ const MyClass *otherClass);

Copy Constructor (continued)

4. WARNING: The follow code produce infinite loop;

   a. MyClass (MyClass other);

5. If a copy constructor is not declared

   a. The compiler will provide a copy constructor implicit
   b. This copy constructor does member-wise copy of the source object
   c. example:

```
MyClass::MyClass(const MyClass& otherClass) : x(otherClass.x),
                                              y(otherClass.y),
                                              z(otherClass.z) {...}
```

6. When do we need to declare a copy constructor?

   a. When member-wise copy is not good enough.
   b. When you need to take a "deep" copy:
      i) The object contains raw pointers
      ii) you do not want to copy the pointer itself; rather, you wish to copy what the pointer points to
      iii) Once the instance own the pointer, the instance is responsible to calling delete at some point.
      If two objects end up calling delete on the same non-NULL pointer it can produce a heap corruption.
   c. Another example:
      i) when you have a reference-counted object.
      boost::shared_ptr<>