VE280 2021SU Final Review Part Ⅰ

Chengsong Zhang continue_revolution@sjtu.edu.cn

L14: Subtypes and Inheritance

Subtype

Subtype relation is an “IS-A” relationship.

For examples, a Swan **is a** Bird, thus a class Swan is a subtype of class Bird. A bird can fly and quake. A swan can also do these.

Reversely, Bird is the **supertype** of the Swan.

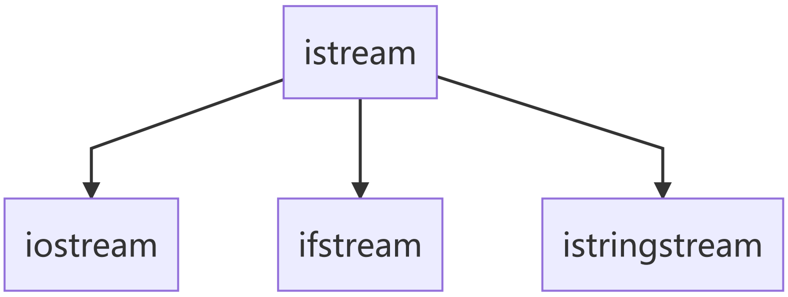
Substitution Principle

If S is a subtype of T or T is a supertype of S, written $S <: T$, then for any instance where an object of type T is expected, we can supply an object of type S.

- Functions written to operate on elements of **the supertype** can also operate on elements of **the subtype.**
- Benefits: code reuse.

父可-》子可

An example would be the input streams of c++:



```
// istream (supertype) supports `>>` operator
int n;
cin >> n;

ifstream inFile;
inFile >> n;

string line; getline(cin, line);
istringstream ss(line);
ss >> n;
```

Distinguish: Substitution vs. Type Coercion

Consider the following examples.

- Example 1: Substitution

  Can we use an `ifstream` where an `istream` is expected? Is there an type conversion happening in this piece of code?

  ```
  void add(istream &source) {
      double n1, n2;
      source >> n1 >> n2;
      cout << n1 + n2;
  }

  int main(){
      ifstream inFile;
      inFile.open("test.in")
      add(inFile);
      inFile.close();
  }
  ```

- Example 2: Type Coercion.

  Can we use an `int` where a `double` is expected? Is there ant type conversion happening in this piece of code?

  ```
  void add(double n1, double n2) {
      cout << n1 + n2;
  }

  int main(){
      int n1 = 1;
      int n2 = 2;          会强制转化为double
      add(n1, n2);
  }
  ```

Creating Subtypes

In an Abstract Data Type, there are three ways to create a subtype from a supertype:

1. Add operations.
2. Strengthen the postconditions of operations
   - Postconditions include:
     - The EFFECTS clause:

       e. g., we print an extra message to `cout` in the new class while returning the same thing
     - The return type:

       e. g., we behave normally under the raw REQUIRES condition and return a special value under the newly allowed REQUIRES condition
3. Weaken the preconditions of operations
   - Preconditions include:

- The REQUIRES clause:

  e. g., we allow negative integers for `insert()`

- The argument type

  e. g., `void operation (PosIntSet s)` -> `void operation (IntSet s)` where `PosIntSet` is a subtype of `IntSet`

Inheritance Mechanism

When a class (called derived, child class or subclass) inherits from another class (base, parent class, or superclass), the derived class is automatically populated with almost **everything from the base class.**

- This includes member variables, functions, types, and even static members.
- The only thing that does not come along is **friendship-ness** **(since a `friend` method is NOT a method of the raw class!!!).**
- We will specifically discuss the inheritance mechanism of constructors and destructors later.

The basic syntax of inheritance is:

```
class Derived : /* access */ Base1, Base2, ... {
private:
    /* Contents of class Derived */
public:
    /* Contents of class Derived */
};
```

When declaring inheritance with access specifiers, the **accessibility of (members that get inherited from the parent class) in the derived classes** are as follows:

子类也不能获取父类的隐私

Inheritance Access Specifier \ Member	private	protected	public
class son：private father **private** 私有继承	inaccessible	转为子类中private private	private
protected	inaccessible	protected	protected
public	inaccessible	protected	public

When you **omit** the inheritance access specifier, the access specifier is set to be `private` by default, similar to the as well.

The following example could help you better understand the above table.

```
class X

{

public:

    int a;

protected:
    int b;

private:

    int c;

};

class Y : public X

{

    // a is public

    // b is protected

    // c is not accessible from Y

};

class Z : protected X

{

    // a is protected

    // b is protected

    // c is not accessible from Z

};

class T : private X

{

    // a is private

    // b is private

    // c is not accessible from T

};
```

Pointer and Reference in Inheritance

From the language perspective, C++ simply trusts the programmer that every subclass is indeed a valid subtype, and therefore we have the following rules.

- Derived class pointer is compatible to base class pointer, i.e., we could run `Player* p1, simplePlayer* p2 = p1`
- Derived class instance is compatible to base class reference, i.e., we could run `countingPlayer s1, Player& s2 = s` 可兼容

父类指针也可以指向子类实例

派生类实例与基类实例兼容

- Derived class instance is compatible to base class instance, i.e., we could run `Jotaro s1, countingPlayer s2 = s1`

In one sentence, whenever you need a base class instance/pointer, you could pass a derived class instance/pointer.

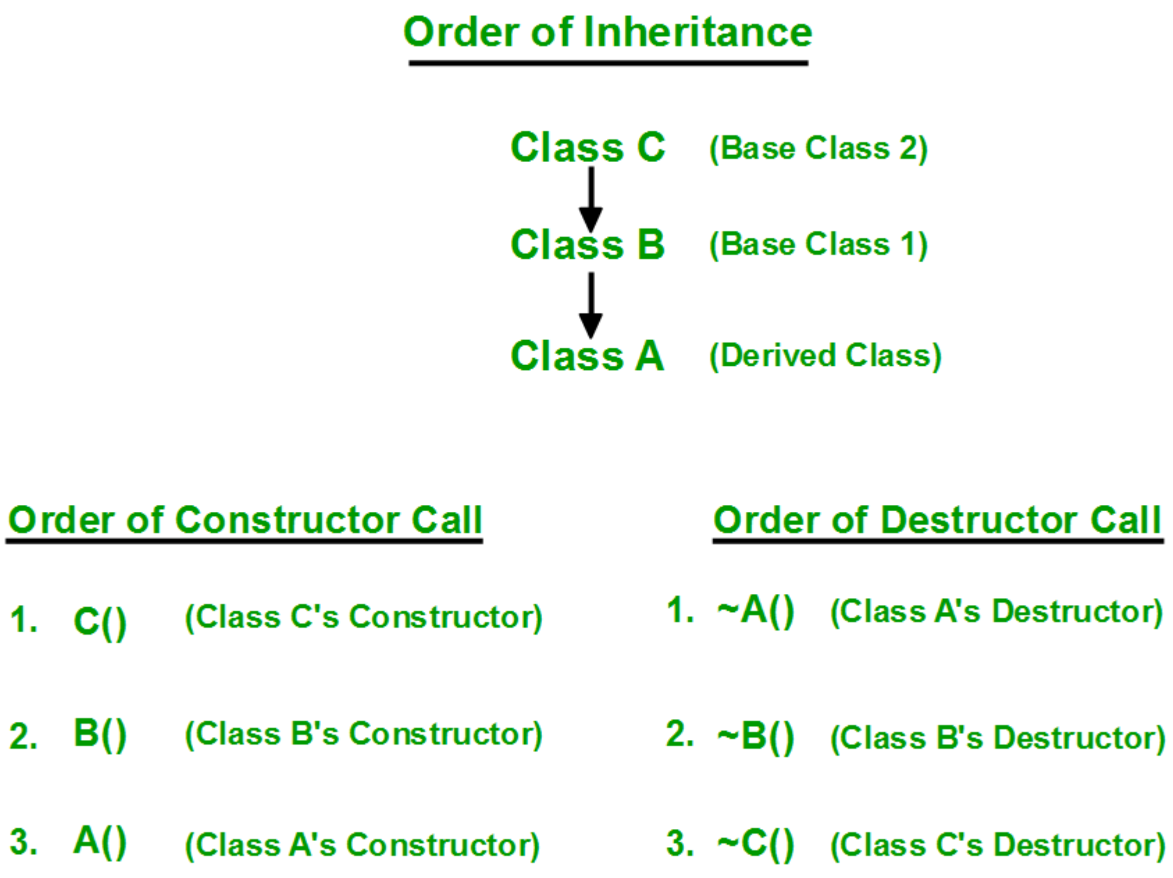
The reverse is generally false. E.g., assigning a base class pointer to derived class pointers needs special casting.

```
Player* p1;countingPlayer* p2;

p1 = p2; // OK       可以把子类指针赋给父类指针
p2 = p1; // ERROR    不能把父类指针赋给子类指针
```

Constructors and Destructors in Inheritance

What would be the order of constructor and destructor call in a inheritance system? A short answer to remember would be:


Order of Inheritance
Class C
(Base Class 2)
Class B
(Base Class 1)
Class A
(Derived Class)
Order of Constructor Call
1. C() (Class C's Constructor)
2. B() (Class B's Constructor)
3. A() (Class A's Constructor)
Order of Destructor Call
1. ~A() (Class A's Destructor)
2. ~B() (Class B's Destructor)
3. ~C() (Class C's Destructor)

Consider the following example:

```
class Parent {
public:
    Parent() { cout << "Parent::Constructor\n"; }
    virtual ~Parent() { cout << "Parent::Destructor\n"; }
};

class Child : public Parent {
public:
    Child() : Parent() { cout << "Child::Constructor\n"; }
    ~Child() override { cout << "Child::Destructor\n"; }
};

class GrandChild : public Child {
public:
    GrandChild() : Child() { cout << "GrandChild::Constructor\n"; }
    ~GrandChild() override { cout << "GrandChild::Destructor\n"; }
};

int main() {
    GrandChild gc;
}
```

The output would be:

```
Parent::Constructor
Child::Constructor
GrandChild::Constructor
GrandChild::Destructor
Child::Destructor
Parent::Destructor
```

Inheritance vs. Subtype

Inheritance is **neither a sufficient nor a necessary condition** of creating subtype in C++.

Non-sufficient: Private inheritance prevents B from being a subtype of A, since **A could not access any members of B**.

私有继承将A中所有public和protected都继承到B的private中变成A不可接触的变量和函数，那么B不是A的子类

```
class A {
    int priv;
};

class B : private A {};
```

Non-necessary: We can create a subtype simply by repeating everything without using inheritance

```
class A {
public:
    void quak(){}
};

class B {
public:
    void quak(){}
    void nop(){}
};
```

可以不继承，复制即可，也能得到

Virtual Functions - Dynamic Polymorphism

Apparent type and Actual type

In default situation, C++ chooses the method to run based on its apparent type.

```
Haruhi h; // Actual type: Haruhi
simplePlayer& s = h; // Apparent type: simplePlayer    这个ref类型是simpleplayer
cout << "Player: " << s.getName() << " bets " << s.bet(8, 5) << endl;
```

If bet function is not virtual, `simplePlayer::bet` will be called and Haruhi will bet 5. However, since she is brave, her bet should always be 10. So you should implement `bet` function as a virtual function. That's why we need `virtual` - to achieve dynamic polymorphism.

`virtual` keyword

A way to tell C++ compiler to choose the **actual type at runtime** before execution.

Using the previous example:

```
class simplePlayer{
    ...
public:
    ...                                         可覆盖的
    virtual int bet(unsigned int bankroll, unsigned int minimum);
    ...
};
```

The above syntax marks insert as a `virtual` function.

Virtual methods are methods **overridable by subclasses.** When a method call is made, if the method you are calling is a virtual function (based on the apparent type), the compiler binds the method according to the **virtual table**. In this way, the function `bet` achieves **dynamic polymorphism**, the ability to change its behavior based on the **actual type** of the argument.

`override` keyword (OPTIONAL, NOT COVERED)

The act of replacing a function is called overriding a base class method. The syntax is as follows.

```
class Haruhi: public simplePlayer{
    ...
public:
    ...
    int bet(unsigned int bankroll, unsigned int minimum) override;
    ...
};
```

`override` cause the compiler to verify if a function is indeed overriding a base class method. If the base class method is not a virtual function, compiler will complain. The keyword is introduced in C++11. **It is considered good programming style to always mark override whenever possible.**

L15: Interfaces and Invariant

Abstract Base Class - Classes as Interfaces

The interface is the **contract for using things of this type (mainly about what I can do with this type).**

The normal C++ class failed to be a perfect interface, it also did not achieve perfect information hiding. It mixes details of the implementation with the definition of the interface. Although the method implementations can be written separately from the class definition and are usually in two separate files. Unfortunately, **the data members still must be part of the class definition.** Since any programmer using your class see that definition, those programmers **know something about the implementation.**

What we prefer is to create an "interface-only" class as a base class, from which an implementation can be derived. Such a base class is called an **Abstract Base Class**, or sometimes a **Virtual Base Class.**

Note:

- An Abstract Base Class **cannot have an real implementation for its methods** since it does not have any data members.
- Also, we **could not create an instance of Abstract Base Class.**

Pure `virtual`

Because there will be no implementation, we need to declare methods in a special way

```
/* Player */ virtual int bet(unsigned int bankroll,
        unsigned int minimum) = 0;;
```

In this case we say the method is pure virtual. 如果有一个或多个纯virtual方法，就说这个class是abstract class.

If a class contains one or more pure virtual methods, we say the class is an **abstract class**. **You only need to have one pure virtual function for a class to be "abstract".** Pure virtual class are also called abstract base classes, or interfaces. It is often that the name abstract base classes starts with a case letter `I` for interface.

As mentioned before, **you can not create an instance of an abstract class.** However, you can always define references and pointers to an abstract class, and use that to refer to an instance of derived class which implement the interface. That is how we get players in P3:

```
static Player player; // ERROR

static Haruhi haruhi;
```

不能创建base class的实例，但是可以创建指针和ref

```
...

Player* get_Player(string &dealerSide, string &playerType , int &ID)
{
    if (dealerSide == "sos") {
        ...
    }
    else if (dealerSide == "sc") {
        if (ID == 1)return &haruhi;
        ...
    }
    else
        ...
}
```

Invariant

An invariant is a set of conditions that must **always evaluate to true** at certain well-defined points; otherwise, the program is incorrect. For ADT, there is so called **representation invariant.**

It describes the conditions that must hold on those members for the representation to correctly implement the abstraction. It must hold immediately before exiting each method of that implementation, including the constructor.

Each method in the class can assume that the invariant is true on entry if the following 2 conditions hold:

- The representation invariant holds **immediately before exiting each method** (including the constructor);
- Each data member is truly private (other functions could not modify the data member).

E.g., `HandValue::count` should always equal the value of hand in `Hand` after calling each method (`addCard`, `discardAll`).

Comprehensive exercise for L13-L15

Here is an comprehensive exercise. What would be the output?

```
#include <iostream>

using namespace std;

class Foo
{
    public:
    void f() { cout << "a"; };
    virtual void g() = 0;
    virtual void c() = 0;
};

class Bar : public Foo
{
    public:
    void f() { cout << "b"; };
    virtual void g() { cout << "c"; };
    void c() { cout << "d"; };
    virtual void h() { cout << "e"; };
};

class Baz : public Bar
{
    public:
    void f() { cout << "f"; };
    virtual void g() { cout << "g"; };
    void c() { cout << "h"; };
    void h() { cout << "i"; };
};

class Qux : public Baz
{
    public:
    void f() { cout << "j"; };
    void h() { cout << "k"; };
};

int main()
{
    Bar bar;    bar.g();
    Baz baz;    baz.h();
    Qux qux;    qux.g();
    Foo &f1 = qux;
    f1.f();    f1.g();    f1.c();
    Bar &b1 = qux;
    b1.f();    b1.c();    b1.h();
    Baz &b2 = qux;
    b2.f();    b2.c();    b2.h();
}
```

只要base class里面的函数是virtual的
那么派生类里对应的函数都是virtual的

如果base class里面函数不是virtual
但是派生类中加上virtual，那么之后派生的类都是

Answer is given below `cigaghbhkfhk` as shown below.

```
int main()
{
    Bar bar;    bar.g(); // c
    Baz baz;    baz.h(); // i
    Qux qux;    qux.g(); // g
    Foo &f1 = qux;
    f1.f();    f1.g();    f1.c(); // a g h
    Bar &b1 = qux;
    b1.f();    b1.c();    b1.h(); // b h k
    Baz &b2 = qux;
    b2.f();    b2.c();    b2.h(); // f h k
}
```

f1是foo类型的ref
b1是bar类型
所以f1找函数从foo 找起，
一直找到最后函数出现的子类

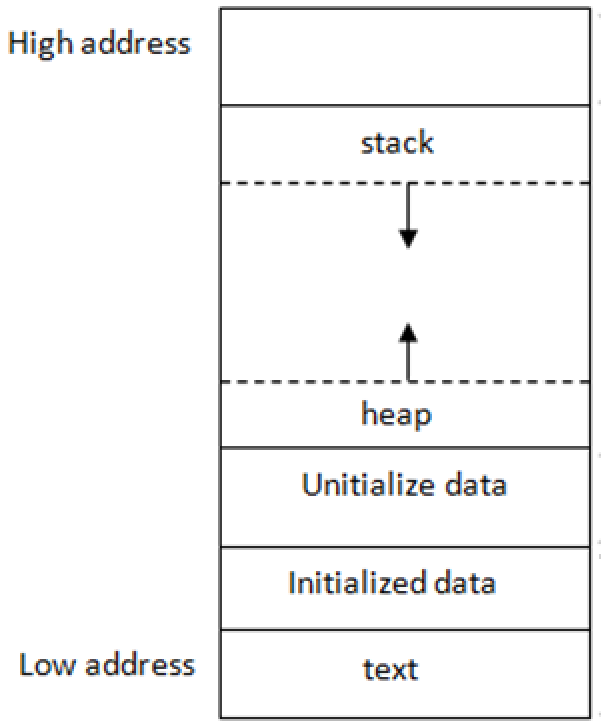
L16: Dynamic Memory Allocation, Overloading, Default Arguments and Destructor

Memory Management (NOT COVERED, MORE IN VE/EECS 370)

Each running program has its own memory layout, separated from other programs. The layout consists of a lot of segments, including:

- `stack`: stores local variables
- `heap`: dynamic memory for programmer to allocate
- `data`: stores global variables, separated into initialized and uninitialized
- `text`: stores the code being executed

In order to pinpoint each memory location in a program's memory, we assign each byte of memory an "address". The addresses go from 0 all the way to the largest possible address, depending on the machine. As the figure below, the `text`, `data`, and `stack` segments have low address numbers, while the `stack` memory has higher addresses.



`new` & `delete`

Static VLA (variable length array) is forbidden (REMEMBER THAT WARNINGS ARE ALSO ERRORS) in C++.

```
int num = 100;
int array[num]; // Error      必须是数字来初始化
```

This leads to:

```
warning: ISO C++ forbids variable length array 'array' [-Wvla]
```

This is where we need dynamic variable length array (whose space is reserved at **runtime**).

```
int num = 100;
int *array = new int[num];
delete [] array;
```

`new`

`new` and `new[]` does the following:

- Allocates space in heap (for one or a number of objects).
- Constructs object in-place (calling the **constructor**).
- Returns the “first” address of the space.

The syntax for `new` operator are very simple.

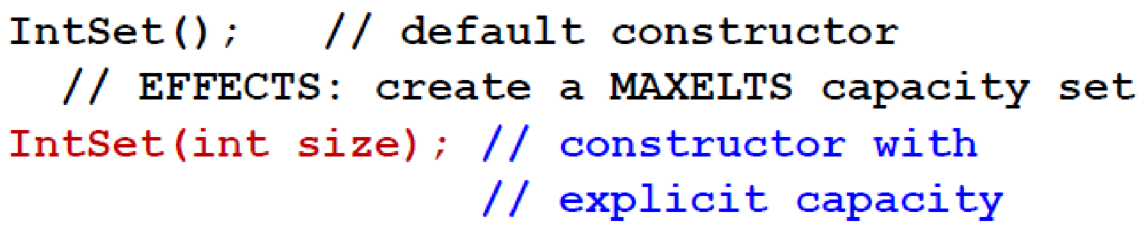
```
Type* obj0 = new Type; // Default construction
Type* obj1 = new Type(); // Default construction
Type* obj2 = new Type(arg1, arg2);
Type* objA0 = new Type[size]; // Default construction for each elt
Type* objA1 = new Type[size](); // Same as above
```

Overloaded Constructor & Default Arguments

We could always create our customized constructor by function overloading.

The overloaded function should

- Has the same name as the default function
- Has a different type signature

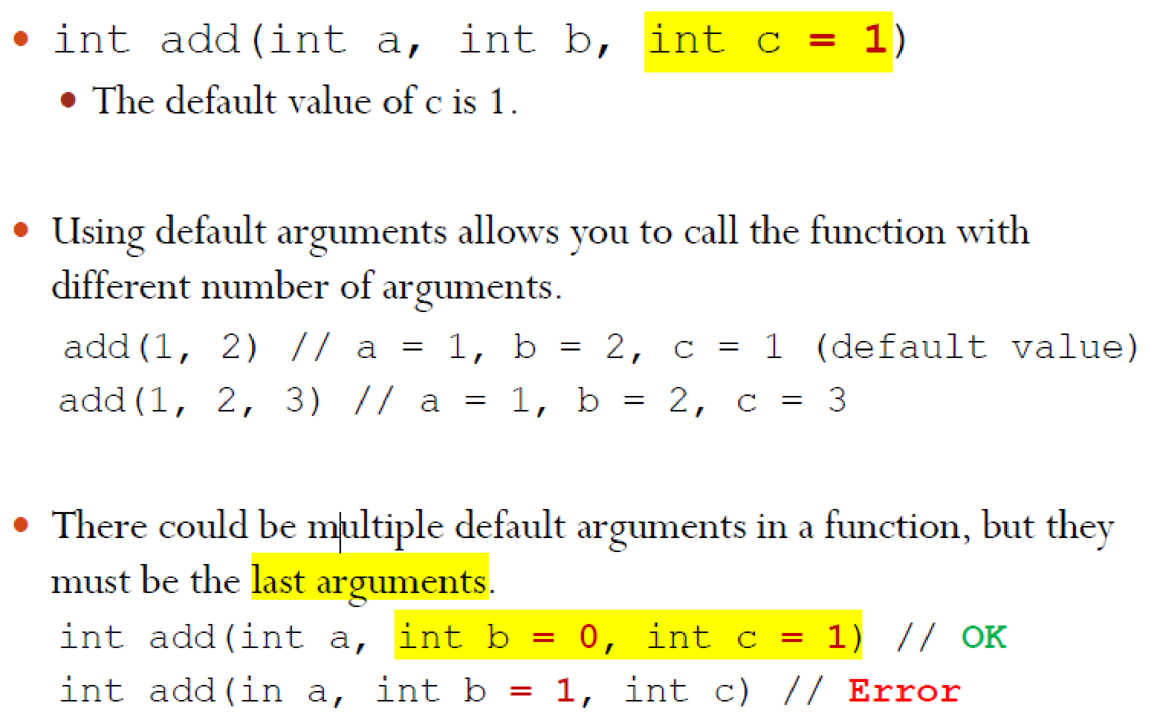


```
IntSet();    // default constructor
  // EFFECTS: create a MAXELTS capacity set
IntSet(int size); // constructor with
                  // explicit capacity
```

Compiler would determine which function to call based on the actual argument counts and types (more in EECS 483).

Also, we could customize **default arguments** for any function.

There could be multiple default arguments in a function, but they must be the **last arguments**.



- `int add(int a, int b, int c = 1)`
  - The default value of c is 1.

- Using default arguments allows you to call the function with different number of arguments.

```
add(1, 2) // a = 1, b = 2, c = 1 (default value)
add(1, 2, 3) // a = 1, b = 2, c = 3
```

- There could be multiple default arguments in a function, but they must be the last arguments.

```
int add(int a, int b = 0, int c = 1) // OK
int add(in a, int b = 1, int c) // Error
```

`delete`

`delete` and `delete[]` releases the objects allocated from `new` and `new[]` respectively. They does the following:

- Destroy the objects (if `[]`, delete each object in the array) being released (by calling the destructor).
- Release the memory/space.

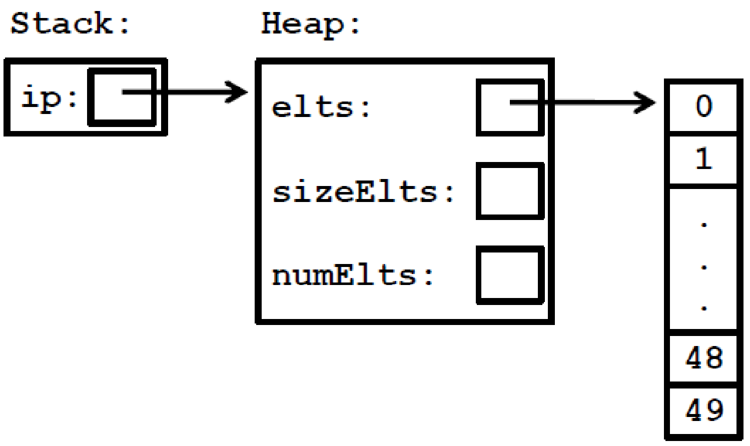
  An example of `ip = new IntSet, delete ip`

```
IntSet *ip = new IntSet(50);
```

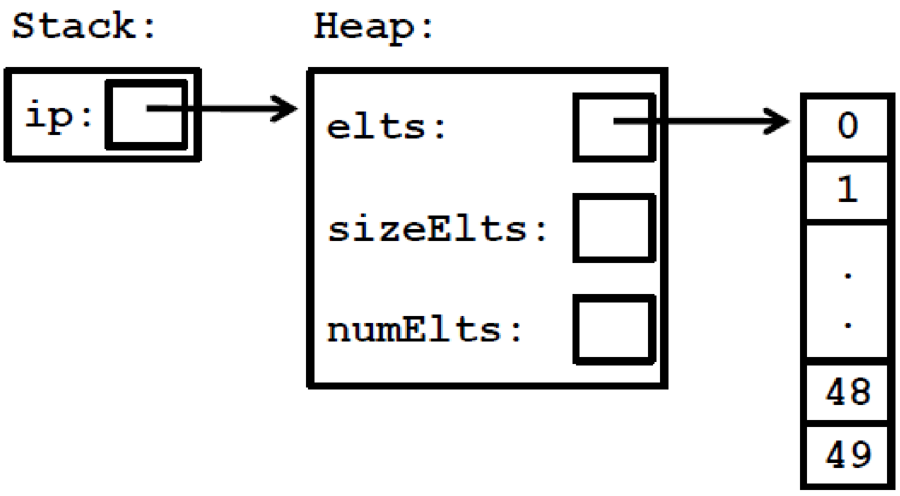
Dynamic Arrays

Dynamic `IntSet` creation

- After the `IntSet` pointer is created, we get:
  - Allocate space to hold the IntSet (a pointer and two integers)
  - Call the constructor on that object (allocates space for the array of 50 integers)



- When you call `delete` on an instance of a class with a destructor
  - **First** the destructor is called (deallocates the array)
  - **Then** the object itself is deleted



We must emphasize that deletion is not idempotent, i.e. `delete` an object **more than once**, or **delete an array** allocated using `new[]` by `delete` instead of `delete[]` cause UB(undefined behavior)!

Destructor

The destructors for any ADTs declared **locally** within a block of code are called automatically when the block ends.

```
Joestar::~Joestar(){
    cout << "Ni Ge Run Da Yo" << endl;
}
int main()
{
    joestar = getPlayer("sos", "counting", 1); // You will get Joseph Joestar
    ...
}
// joestar will be destroyed by calling its destructor
// The program will output "Ni Ge Run Da Yo".
```

Destructor of dynamic ADT will be called using `delete`.

An effective destructors should:

- Be named as `~<ClassName>`
- Takes no argument and returns nothing (not even `void`)
- If one expect the class to be inherited the destructor should be declared as `virtual` (NOT COVERED THIS TERM, OPTIONAL)
- Release resource allocated only **in this class**, do not release **base class** resources!!! (NOT COVERED THIS TERM, OPTIONAL)

An example for `Node` in Lab8:

```
Node::~Node() {
    ...
    delete[] children;
}
```

Memory Leaks

If an object is allocated, but not released after the program is done with it, the system would **assume that the resource is still being used**, but the program will never use it. Thus this resource is “leaked”, i.e. **no longer available for using.** In our case the leaked resource is memory.

Note that memory leaks will not be caught by the compilers!

`valgrind` is a tool that can help you detect potential memory leaks. It actually looks for for all sorts of memory related problems, including:

- Memory Leaks
- Invalid accesses
  - Array out-of-range
  - Use of freed memory

- Double free problems

Consider the following examples.

1. Memory Leaks.

```
int *p1 = new int(1);
int *p2 = new int(2);
p1 = p2;
// There is no way to release the memory occupied by "1"
```

2. Invalid Access

```
int arr = new int[5];
cout << arr[5] << endl;

delete arr[];
cout << arr[0] << endl;
```

3. Double free

```
int *p1 = new int(1);
int *p2 = new int(2);
p1 = p2;
delete p1;
delete p2;
```