Introducción a C++

Algoritmos y Estructuras de Datos II

DC-FCEyN-UBA

26 de agosto de 2015

Estructura de un programa C++

```
1       // Archivo hola.cpp
2
3       #include <iostream>
4
5       using namespace std;
6
7       int main( ) {
8
9           cout << "Hola Mundo" << endl;
10
11          for (int i = 0; i <= 10; i++) {
12      cout << i << endl;
13  }
14
15  for (int i = 0, j = 0; i <= 10; i++, j++) {
16      cout << i+j << endl;
17  }
18
19          return 0;
20      }
```

Memoria dinámica

Tiempo de vida de una variable

Toda variable tiene un tiempo de vida, **comienza** cuando se declara y **finaliza** cuando se sale del *scope* que la declaró.

```
int h = 4;
for(int i=0;i<10;i++){
      int g = 75;
}
cout << g << endl; //esto esta bien??
```

Tiempo de vida de una variable

Toda variable tiene un tiempo de vida, **comienza** cuando se declara y **finaliza** cuando se sale del *scope* que la declaró.

```
int h = 4;
for(int i=0;i<10;i++){
    double h = 29; //compila??
}
cout << h << endl;
```

El compilador se encarga de usar la pila o *stack* que el SO provee para almacenar las variables locales (y los parámetros de las funciones).

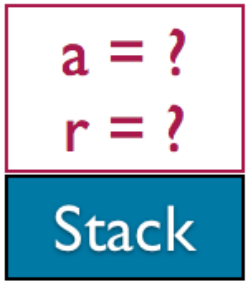
Veamos un ejemplo!

Variables en el stack (1/10)

```
int sum(int x, int y) {
   int r = x + y;
   return r;
}

int cuad(int x) {
   int c = sum(x, x);
   c = sum(c, c);
   return c;
}

int main() {
   int a;
   cin >> a;
   int r = cuad(a);
   // ...
   return 0;
}
```



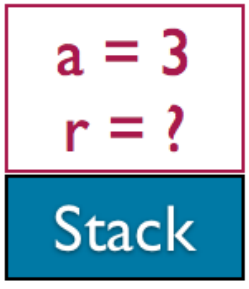
Variables en el stack (2/10)

```
int sum(int x, int y) {
   int r = x + y;
   return r;
}

int cuad(int x) {
   int c = sum(x, x);
   c = sum(c, c);
   return c;
}

int main() {
   int a;
   cin >> a;
   int r = cuad(a);
   // ...
   return 0;
}
```

cin >> a;



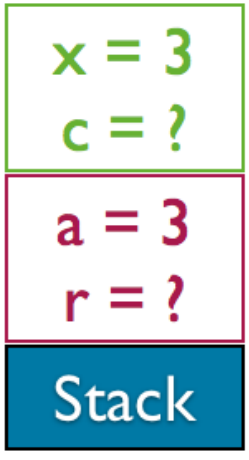
Variables en el stack (3/10)

```
int sum(int x, int y) {
   int r = x + y;
   return r;
}

int cuad(int x) {
   int c = sum(x, x);
   c = sum(c, c);
   return c;
}

int main() {
   int a;
   cin >> a;
   int r = cuad(a);
   // ...
   return 0;
}
```

cuad(a)



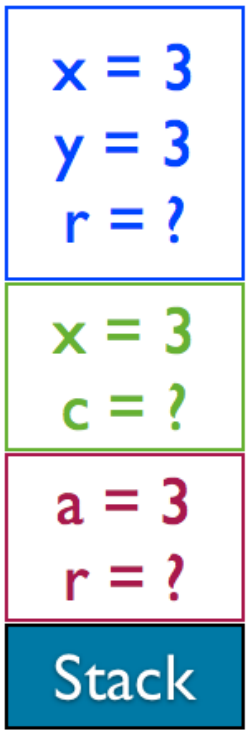
Variables en el stack (4/10)

```
int sum(int x, int y) {
   int r = x + y;
   return r;
}

int cuad(int x) {
   int c = sum(x, x);
   c = sum(c, c);
   return c;
}

int main() {
   int a;
   cin >> a;
   int r = cuad(a);
   // ...
   return 0;
}
```

sum(x, x);



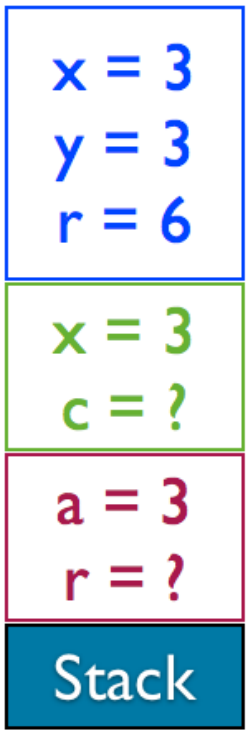
Variables en el stack (5/10)

```
int sum(int x, int y) {
   int r = x + y;
   return r;
}

int cuad(int x) {
   int c = sum(x, x);
   c = sum(c, c);
   return c;
}

int main() {
   int a;
   cin >> a;
   int r = cuad(a);
   // ...
   return 0;
}
```

r = x + y;



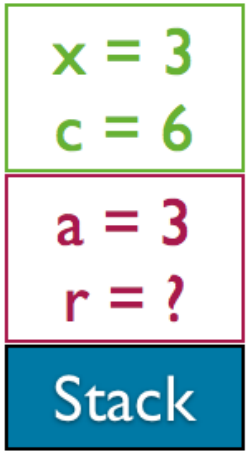
Variables en el stack (6/10)

```
int sum(int x, int y) {
   int r = x + y;
   return r;
}

int cuad(int x) {
   int c = sum(x, x);
   c = sum(c, c);
   return c;
}

int main() {
   int a;
   cin >> a;
   int r = cuad(a);
   // ...
   return 0;
}
```

```
return r;
int c = ...
```



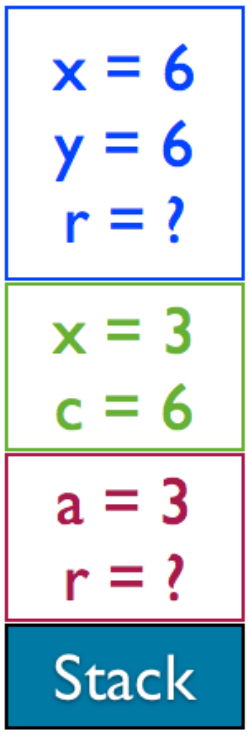
Variables en el stack (7/10)

```
int sum(int x, int y) {
   int r = x + y;
   return r;
}

int cuad(int x) {
   int c = sum(x, x);
   c = sum(c, c);
   return c;
}

int main() {
   int a;
   cin >> a;
   int r = cuad(a);
   // ...
   return 0;
}
```

sum(c,c);



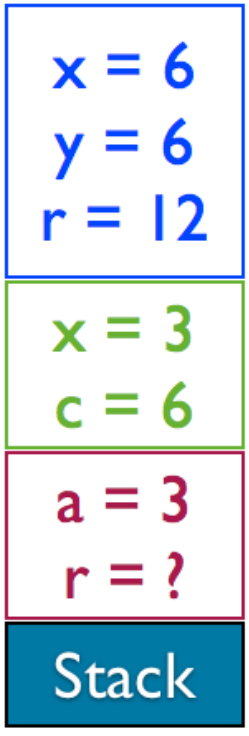
Variables en el stack (8/10)

```
int sum(int x, int y) {
   int r = x + y;
   return r;
}

int cuad(int x) {
   int c = sum(x, x);
   c = sum(c, c);
   return c;
}

int main() {
   int a;
   cin >> a;
   int r = cuad(a);
   // ...
   return 0;
}
```

r = x + y;



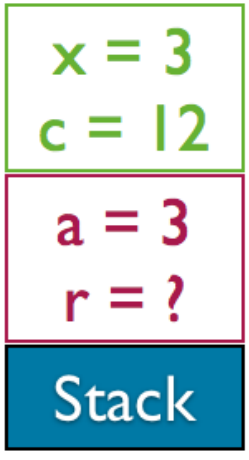
Variables en el stack (9/10)

```
int sum(int x, int y) {
   int r = x + y;
   return r;
}

int cuad(int x) {
   int c = sum(x, x);
   c = sum(c, c);
   return c;
}

int main() {
   int a;
   cin >> a;
   int r = cuad(a);
   // ...
   return 0;
}
```

```
return r;
c = ...
```



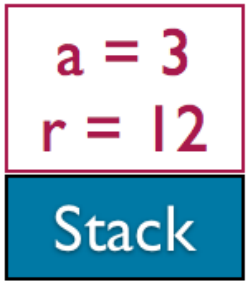
Variables en el stack (10/10)

```
int sum(int x, int y) {
   int r = x + y;
   return r;
}

int cuad(int x) {
   int c = sum(x, x);
   c = sum(c, c);
   return c;
}

int main() {
   int a;
   cin >> a;
   int r = cuad(a);
   // ...
   return 0;
}
```

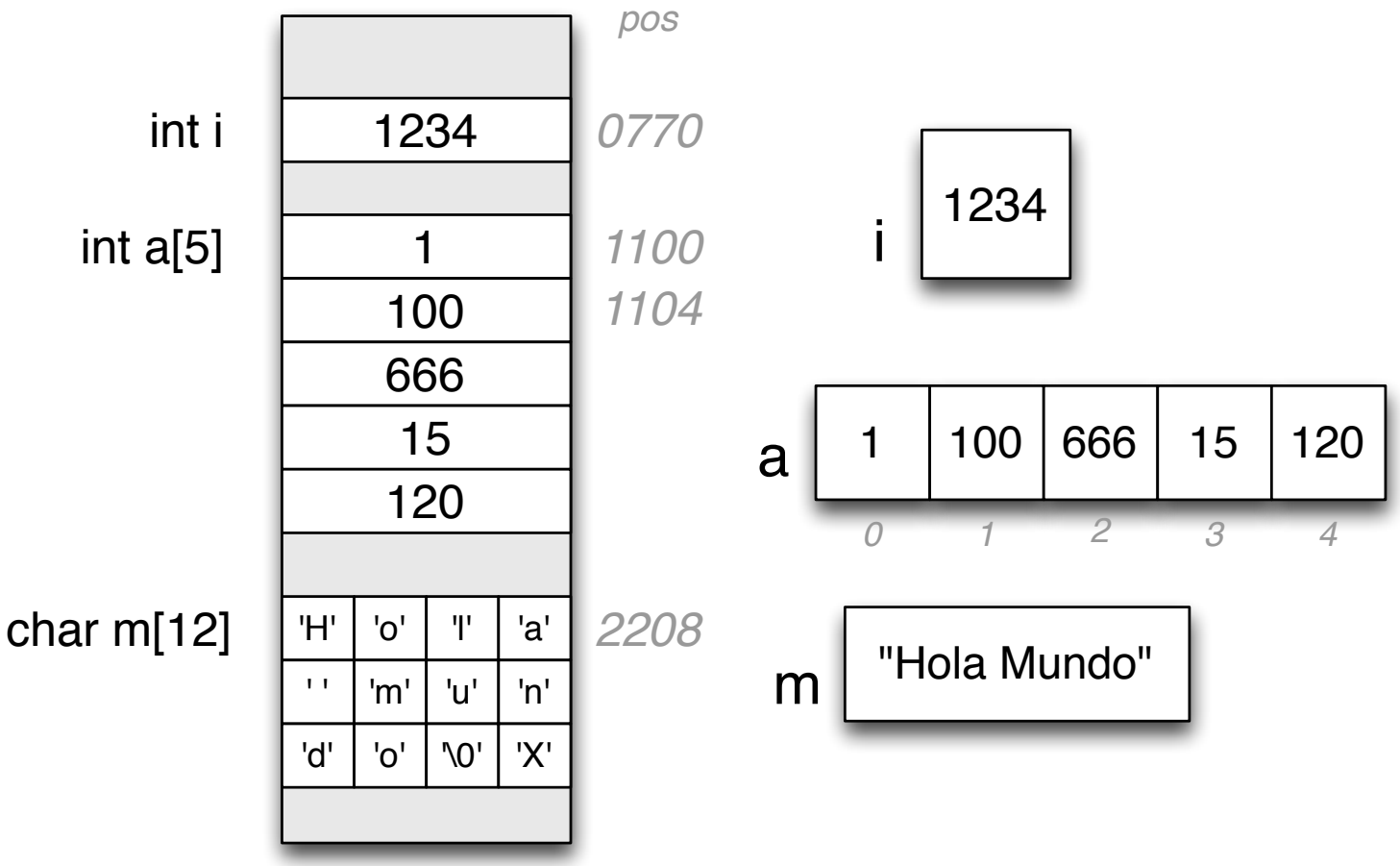
return c;
r = ..



Modelo de memoria: Abstrayendo

- Para C++, la memoria es simplemente un *array de bytes*
- Cada variable ocupa una o más posiciones del array según su tamaño
- El tamaño depende del tipo, del compilador, y de la arquitectura

Veamos un mapa



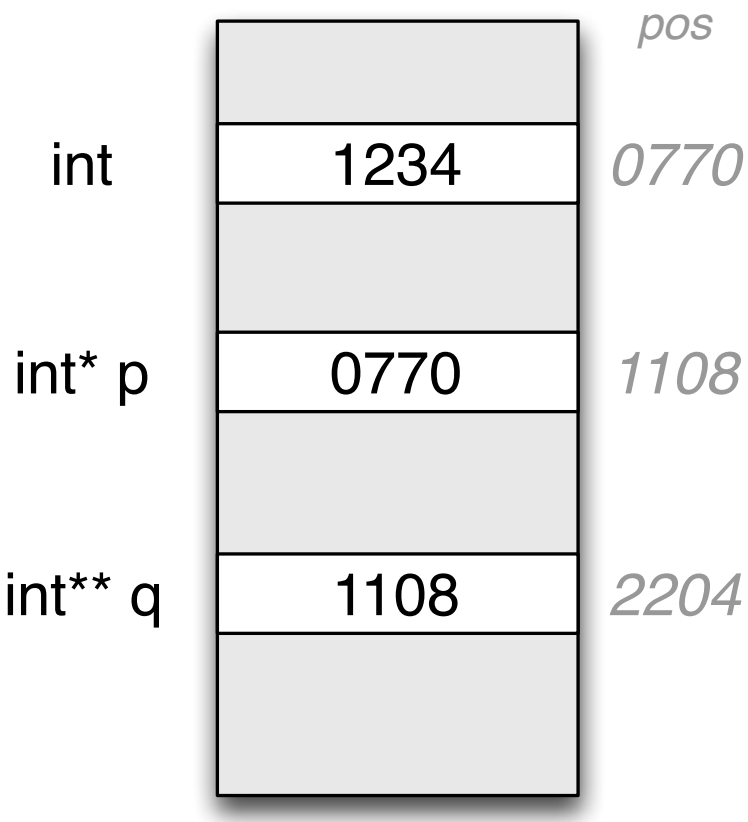
Punteros

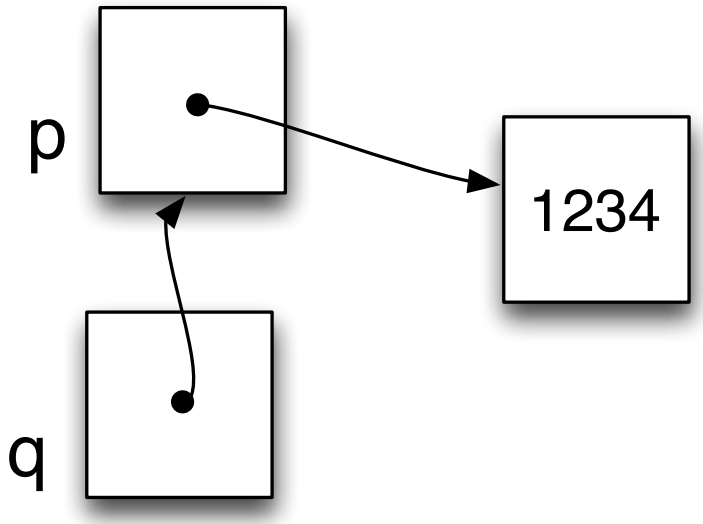
- Dado el tipo T, el tipo T* se denomina “puntero a T”
- Valor (numérico): *una dirección de memoria*
- El valor de memoria “0” (NULL) está reservado como “invalido”
- Su tipo nos dice a qué tipo de objeto apunta
- Se puede tener T**, T***, . . . (puntero a puntero a T, puntero a puntero a puntero a T,. . .)

Repitan conmigo: FLECHAS

- Un puntero se ve más claramente como una flecha ($\rightarrow$) que apunta a un objeto en algún lugar de la memoria
- Varios punteros pueden apuntar a lo mismo generando *aliasing* de punteros. (¡¡no borres más de una vez lo mismo!!)
- Un puntero declarado y no inicializado apunta a cualquier cosa (peligro!)

¡Veo los punteros!



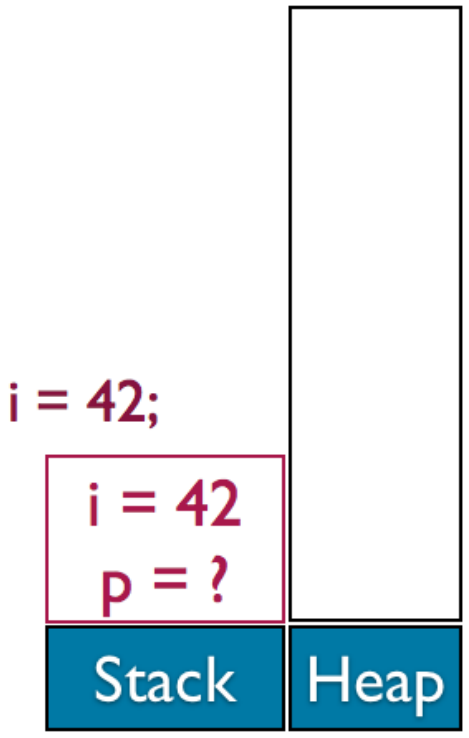


Operando con punteros

- *new T*: devuelve un T* apuntando a un nuevo objeto de tipo T (alojado en el *heap*)
- *new T[N]*: devuelve un T* apuntando a un array de N elementos de tipo T (alojados "consecutivamente" en el *heap*)
- *p[i]*: devuelve el iésimo elemento de un array creado dinámicamente con *new T[N]* y **p* devuelve la primera posición.
- *delete p / delete [] p*: borra lo que está siendo apuntado por el puntero p (¡sólo si lo que apunta fue creado con new!). delete[] borra arrays dinámicos. No confundirlos!!!
- **p*: devuelve el **valor** apuntado por p
- *p→(...)*: equivalente a "(*p).(...)". Para usar con clases/structs.
- *&v*: Sea v **variable** de tipo T, &v es un T* con la dirección de memoria donde se aloja v

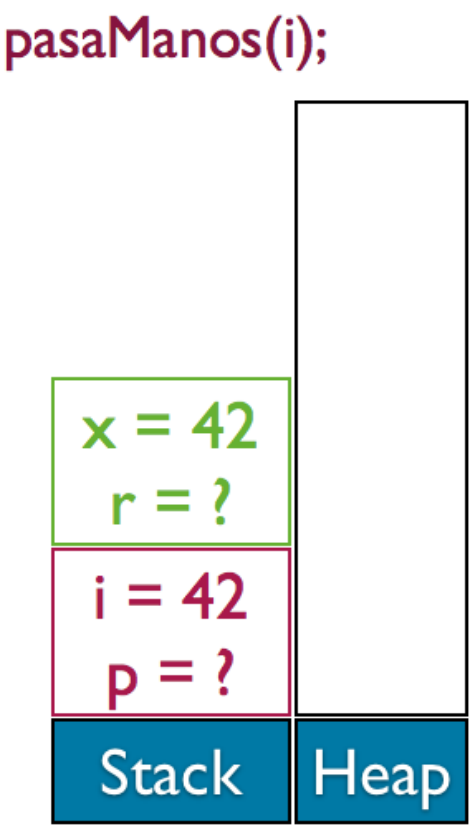
Variables en el heap (1/9)

```
int* crearOtro(int v) {
   int* res = new int;
   *res = v
   return res;
}
int* pasaManos(int x) {
   int* r = crearOtro(x);
   return r;
}
int main() {
   int i = 42;
   int* p;
   p = pasaManos(i);
   *p = *p + 1;
   delete p;
   return 0;
}
```



Variables en el heap (2/9)

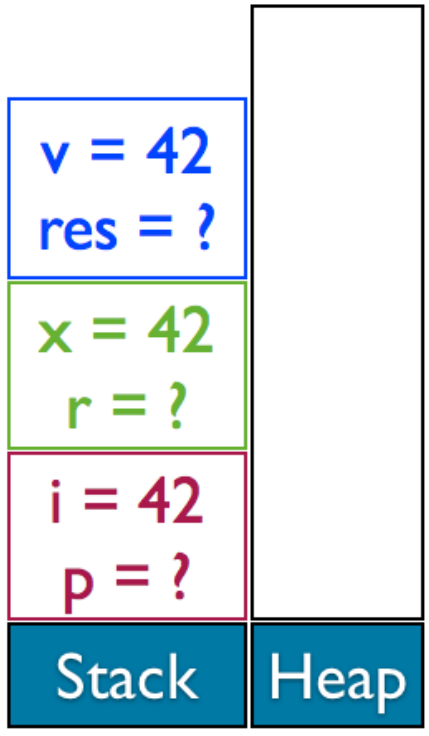
```
int* crearOtro(int v) {
   int* res = new int;
   *res = v
   return res;
}
int* pasaManos(int x) {
   int* r = crearOtro(x);
   return r;
}
int main() {
   int i = 42;
   int* p;
   p = pasaManos(i);
   *p = *p + 1;
   delete p;
   return 0;
}
```



Variables en el heap (3/9)

```
int* crearOtro(int v) {
   int* res = new int;
   *res = v
   return res;
}
int* pasaManos(int x) {
   int* r = crearOtro(x);
   return r;
}
int main() {
   int i = 42;
   int* p;
   p = pasaManos(i);
   *p = *p + 1;
   delete p;
   return 0;
}
```

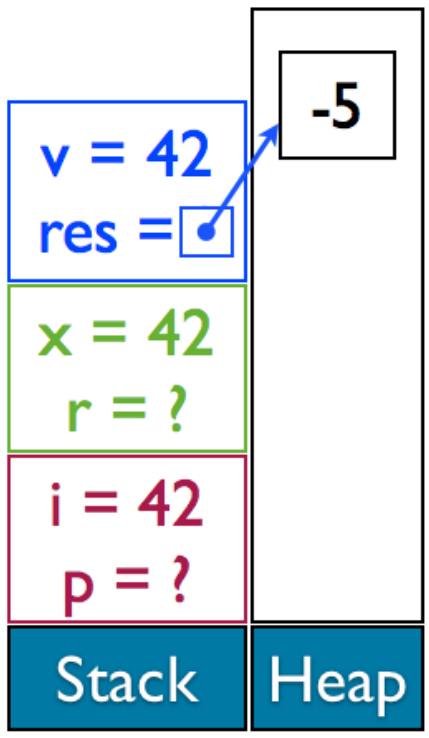
crearOtro(x);



Variables en el heap (4/9)

```
int* crearOtro(int v) {
   int* res = new int;
   *res = v
   return res;
}
int* pasaManos(int x) {
   int* r = crearOtro(x);
   return r;
}
int main() {
   int i = 42;
   int* p;
   p = pasaManos(i);
   *p = *p + 1;
   delete p;
   return 0;
}
```

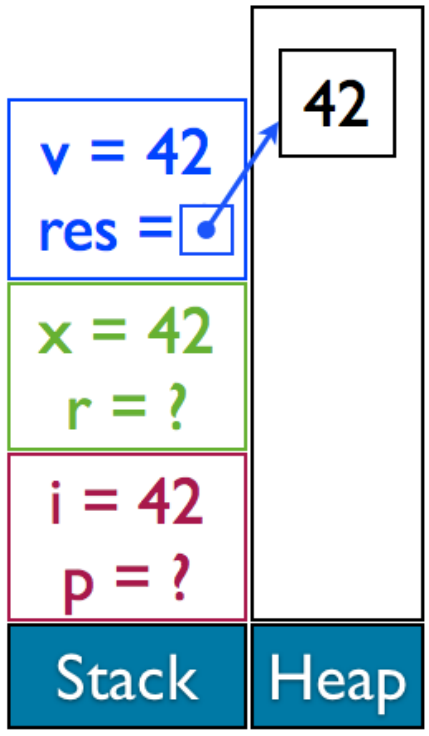
res = new int;



Variables en el heap (5/9)

```
int* crearOtro(int v) {
   int* res = new int;
   *res = v
   return res;
}
int* pasaManos(int x) {
   int* r = crearOtro(x);
   return r;
}
int main() {
   int i = 42;
   int* p;
   p = pasaManos(i);
   *p = *p + 1;
   delete p;
   return 0;
}
```

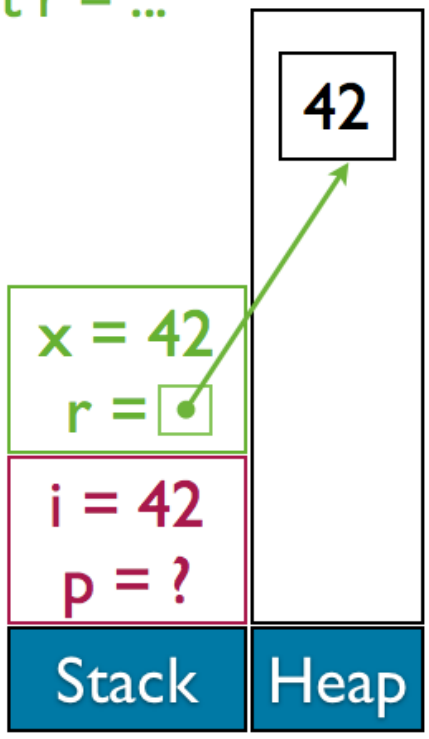
*res = v;



Variables en el heap (6/9)

```
int* crearOtro(int v) {
   int* res = new int;
   *res = v
   return res;
}
int* pasaManos(int x) {
   int* r = crearOtro(x);
   return r;
}
int main() {
   int i = 42;
   int* p;
   p = pasaManos(i);
   *p = *p + 1;
   delete p;
   return 0;
}
```

return res;
int r = ...

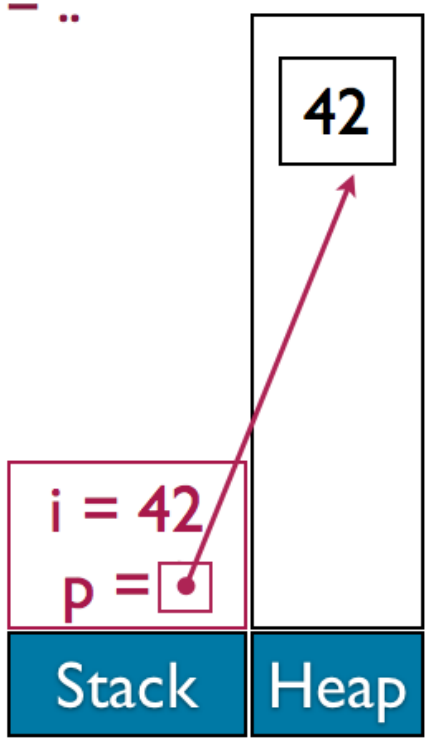


Variables en el heap (7/9)

```
int* crearOtro(int v) {
   int* res = new int;
   *res = v
   return res;
}
int* pasaManos(int x) {
   int* r = crearOtro(x);
   return r;
}
int main() {
   int i = 42;
   int* p;
   p = pasaManos(i);
   *p = *p + 1;
   delete p;
   return 0;
}
```

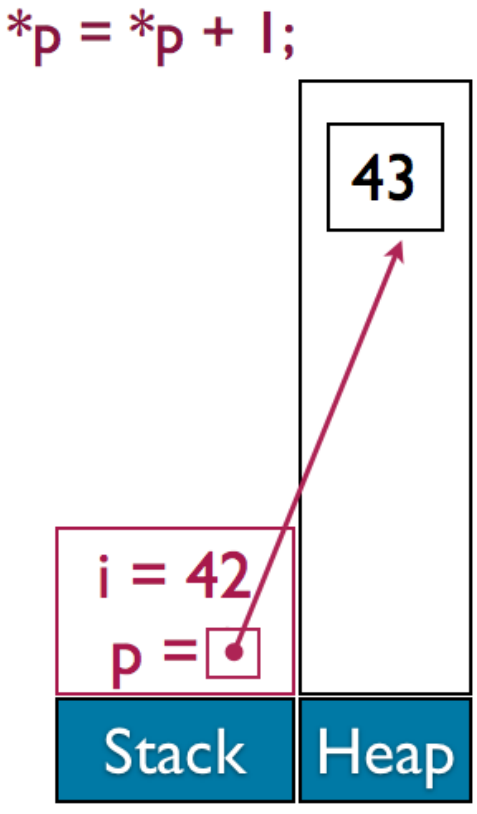
return r;

p = ..



Variables en el heap (8/9)

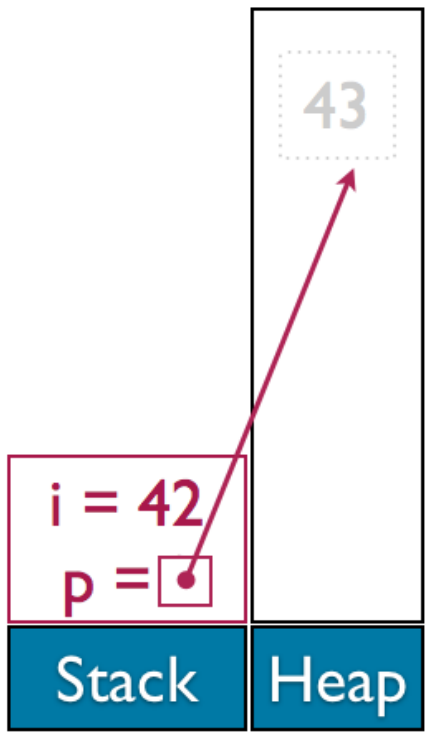
```
int* crearOtro(int v) {
   int* res = new int;
   *res = v
   return res;
}
int* pasaManos(int x) {
   int* r = crearOtro(x);
   return r;
}
int main() {
   int i = 42;
   int* p;
   p = pasaManos(i);
   *p = *p + 1;
   delete p;
   return 0;
}
```



Variables en el heap (9/9)

```
int* crearOtro(int v) {
   int* res = new int;
   *res = v
   return res;
}
int* pasaManos(int x) {
   int* r = crearOtro(x);
   return r;
}
int main() {
   int i = 42;
   int* p;
   p = pasaManos(i);
   *p = *p + 1;
   delete p;
   return 0;
}
```

delete p;



Ejemplos

```
// p: en el stack, pk lo va a apuntar
void funcAburrida(int N) {
  long k = 1234;
  long * pk, * pk2;           // pk, pk2 -> ???
  pk = &k;                    // pk -> k
  cout << k << "==" << *pk << "==" << *&k << endl;
  pk = new long;      // nuevo long en heap
  *pk = 10;
  (*pk)++;            // ese long ahora vale 11
  pk2 = new long[N]; // nuevo array de long en heap
  cout << ( *pk2 == pk2[0] ) << endl;
  pk2[20] = 2932;     // asigno posicion 20 en pk2
  delete pk;
  delete [] pk2;  // borro toda la memoria del array
}
```

Ejemplos (2)

```
struct punto {int x; int y;};

void funcMasAburrida() {
  punto * p = new punto;
  cout << p->x << '==' (*p).x << endl;
}
```

Referencias

- "Punteros" disfrazados de corderos (*igual son preferibles*)
- Dado el tipo T, T& es el tipo referencia a un objeto de tipo T
- También "flecha", pero que se usa exactamente igual que el objeto original
- Más seguros: la dirección de memoria está "escondida"
- Una referencia debe inicializarse con un objeto existente al declararse (o no compila): no pueden haber referencias apuntando a algo inválido o a NULL (*pero recordemos que se implementan con punteros...*)
- *Pero por qué las hay?* Porque en el fondo son punteros... !!!

Ejemplos (2)

```
void funcMortalmenteAburrida() {
  int olerante = 10;
  int & loco = olerante;
  int & errorDeCompilacion; //No inicializado!
  olerante += 10;
  loco += 10; // es una ref, pero se usa igual
  // cuanto valen ahora?
}
long & funcMortal() {
  long aniza = 10;
  return aniza;      // boom... por que?
}
```

Tiempo de vida de una variable (bis)

Toda variable tiene un tiempo de vida, **comienza** cuando se declara y **finaliza** cuando se sale del *scope* que la declaró.

Memoria dinámica

La memoria pedida con *new* tiene un tiempo de vida que **comienza** cuando se llama a *new* y **termina** cuando se le hace *delete* al puntero (o termina el programa en ejecución (proceso) correspondiente).

Atención

Notar que una variable de tipo puntero muere cuando finaliza su scope pero no así la memoria a la que apunta. Cuidado con los leaks!

Stack Vs. Heap: conclusiones y diferencias

- Dentro de un programa, el stack es un área de memoria de tamaño fijo y limitado, mientras que el heap es variable e "ilimitado".
- El stack lo usa y administra el compilador mediante el pasaje a assembler y no se puede controlar desde adentro del programa.
- El heap lo administra el programador mediante llamadas a funciones de la librería standar de C/C++ (malloc/free, new/delete).
- El stack y el heap son espacios de memoria disjuntos. No comparten direcciones de memoria. Atención al guardar variables del stack EN el heap y viceversa.

Reflexiones sobre punteros y memoria dinámica

- Sin punteros y memoria dinámica no podemos crear más cosas que las que declaramos como variables.
- En C++, sin punteros son imposibles los tipos de datos recursivos
- ¿Qué otras estructuras no se podrían implementar sin punteros?

Estructuras de datos, para qué

Ejemplo: secuencia

- Queremos una secuencia de tamaño dinámico, con operaciones: nueva, agregarAtras/Adelante, iésimo, longitud, etc... ¿Cómo hacemos?
- ¿Cómo podemos implementar la secuencia con arreglos?
- Solución: cadena de nodos de longitud variable.
- Sin usar memoria dinámica no alcanza... ¿por qué?

Clases

Clases

Una clase es una estructura de datos que puede contener datos y funciones. Además, las clases permiten definir la visibilidad de sus miembros.

```
class Rectangle {
    private:
        int x, y;
    public:
        void set(int x, int y);          // Declaracion
        int area() { return x * y; }     // Ambas
};

void Rectangle::set(int a, int b) {      // Definicion
    this->x = a;
    this->y = b;
}
```

El *keyword* `this` es un valor especial con un puntero a la representación de la clase, este valor sólo tiene sentido utilizarlo dentro de la definición de un método de la clase.

Clases

¿Cómo se invocan los métodos de una clase?

```
int main(int argc, char** argv) {
    Rectangle r;
    r.set(10, 20);
    Rectangle* pr = &r;
    cout << pr->area() << endl;
}
```

Al igual que para acceder a los miembros de una estructura el ‘.’ y el operador ‘->’ sirven para invocar los métodos sobre una instancia de la clase.

Clases

Sobre una instancia constante sólo se pueden invocar métodos que garanticen que el estado interno de la clase no será alterado. Para ello se utiliza el *keyword* `const` al final de la declaración de los métodos constantes.

```
class Rectangle {
    private:
        int x, y;
    public:
        void set(int x, int y);
        int area() const { return x * y; }
};
```

Constructor, destructor y operador de asignación

Por default C++ nos provee de un mecanismo de construcción y destrucción, y un operador de asignación. Esos comportamientos pueden ser reemplazados por otros definidos por el programador.

```
class Rectangle {
    int x, y; // por default private

    public:

        Rectangle(int a, int b) : x(a), y(b) {}

        ~Rectangle() {}

        Rectangle& operator=(const Rectangle& other);

};

Rectangle& Rectangle::operator=(const Rectangle& other) {
    this->x = other.x;
    this->y = other.y;
}
```

Constructor, destructor y operador de asignación

En el ejemplo se utiliza una lista de inicializadores para asignar los valores de 'x' e 'y' en la construcción.

```
Rectangle(int a, int b) : x(a), y(b) {}
```

El destructor vacío es equivalente al default. No usar new nos garantiza no perder memoria, y por lo tanto, no definir el destructor. Liberar con delete la memoria que pedimos con new, únicamente de nuestra clase y **no** en otras.

```
~Rectangle() {}
```

La implementación del operador de asignación es equivalente a la default. Notar que si bien 'x' e 'y' son privados la implementación de la operación puede accederlos ya que se declara dentro de la clase.

```
Rectangle& Rectangle::operator=(const Rectangle& other) {
    this->x = other.x;
    this->y = other.y;
}
```

Constructor, destructor y operador de asignación

Otra funcionalidad provista por default es la construcción por copia. Ésta se utiliza cada vez que se necesita copiar un valor del tipo definido por la clase (por ejemplo al pasar por parámetro o retornar en una función).

```
class Rectangle {
    int x, y;

    public:

        Rectangle(const Rectangle& other) :
            x(other.x), y(other.y) {}

        ...
};
```

El definido en el ejemplo equivale al comportamiento por default. Recordar que si una clase tiene una representación extensa el costo del copiado puede ser muy elevado.

Pasaje de parámetros por referencia o por copia

```
class Grandota {
  Grandota(){ a = new int[100000];}
  Grandota(const Grandota& gr) {
    this->a = new int[100000];
    for (int i=0; i < 100000, i++) {
      this->a[i] = gr.a[i];
    }
  }
  ~Grandota(){ delete [] a; }

  int* a;
};

void funPesada(Grandota g); //que recibe la funcion?
void funLiviana(Grandota& g); //y aca?
void funLivianaYSegura(const Grandota& g); //y aca?
```

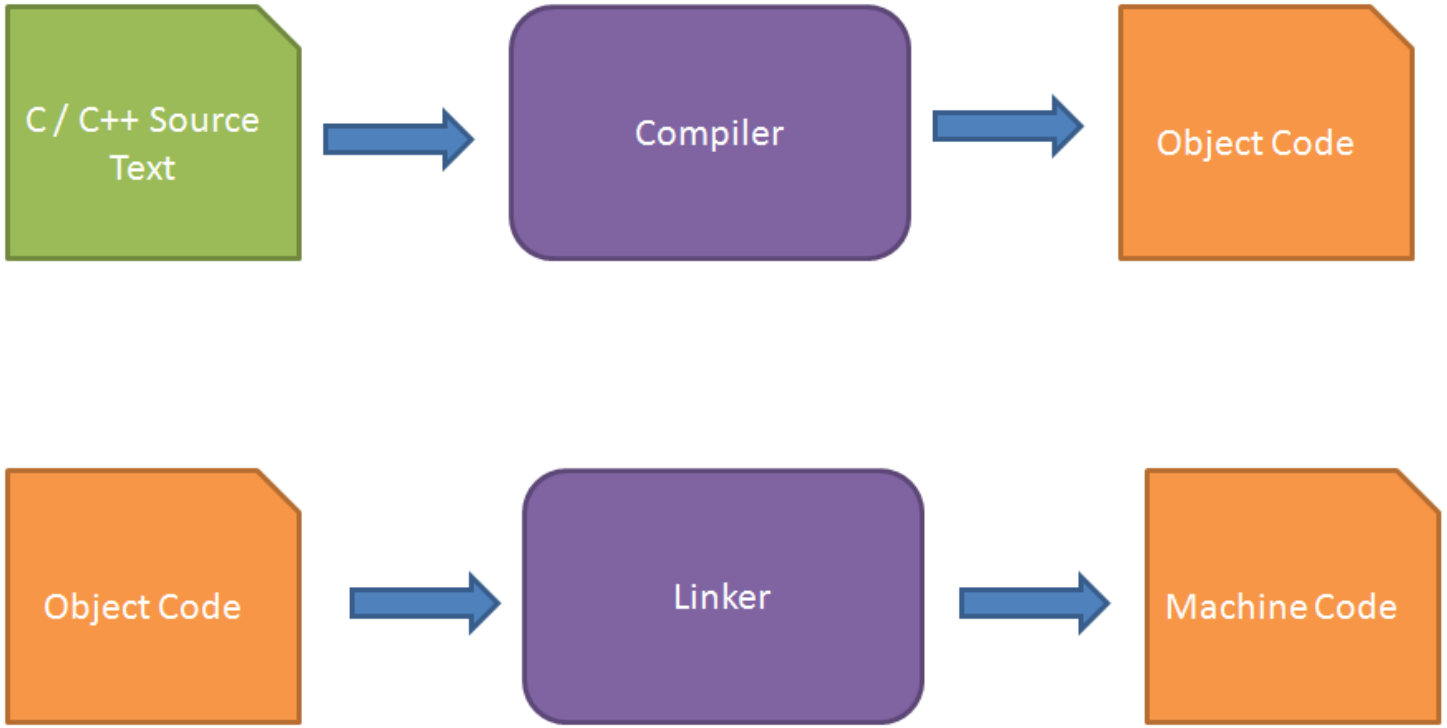
Consejos de memoria al definir una clase

Parámetros por copia o por referencia?

- Prestar especial atención a cuando se pasan parámetros por copia. Puede tener impacto en el *orden de complejidad* del método.
- Si mi clase no tiene definido el constructor por copia puedo estar metiendo la pata bien feo (pensar en el caso del ejemplo anterior).
- Pasar por *const copia* no tiene sentido!
- En algunos casos es válido retornar un parámetro por copia (en qué casos?).
- No definir siempre todos los parámetros por copia, usar referencias y referencias constantes a const-ciencia.

Compilación, linkeo y testing

Compilación y linkeo I



El compiler compila cada archivo por separado y genera un .o para cada uno.
El linker toma todos los .o y genera un solo ejecutable.

Compilación y linkeo II

¿Qué es el compilador ?
El compilador es un programa que toma como entrada archivos con código fuente, cada uno representa una unidad de compilación, y los traduce a código objeto (lenguaje de máquina).

```
g++ -c hola.cpp -o hola.o
```

¿Qué es el linker ?
El linker es un programa que toma archivos con código objeto y los une en un único programa ejecutable (cuyo punto de entrada es la función main).

```
g++ hola.o -o hola
```

Alternativamente pueden hacerse ambos pasos simultáneamente:

```
g++ hola.cpp -o hola
```

Directivas del preprocesador

Se pueden utilizar las siguientes directivas que son resueltas en tiempo de compilación:

- `#define` que define una constante o una macro
- `#undef` elimina una definición anterior
- `#if #else #elif #endif` que permiten tomar decisiones en tiempo de compilación
- `#ifdef #ifndef defined` que permiten verificar si una constante fue definida previamente

Adicionalmente se pueden definir constantes desde la línea de comando del compilador con la opción `-D`.

Directivas del preprocesador

Es recomendable no abusar del preprocesador, ya que al nivel de la materia no es necesario. Sin embargo, encontrarán la necesidad de usarlo para no incluir múltiples veces los mismos archivos de encabezado (lo que lleva a un error de compilación).

```
// Archivo Rectangle.h
#ifndef RECTANGLE_H
#define RECTANGLE_H
    ...
#endif
```

Makefiles e IDEs

¿Porqué compilar y luego linkear ?
La ventaja de hacer la compilación de cada unidad de compilación y luego el *linking* es que no es necesario recompilar todo los archivos con código fuente si sólo cambia un subconjunto estricto.

¿Qué son los makefiles ?
Los *Makefiles* son archivos con instrucciones para el proceso de compilación. La herramienta que interpreta estos archivos se llama *make*. Make detecta automáticamente si un archivo fue modificado desde la última compilación. Muchos proyectos *open source* distribuyen la configuración para la compilación en makefiles u otros formatos similares.

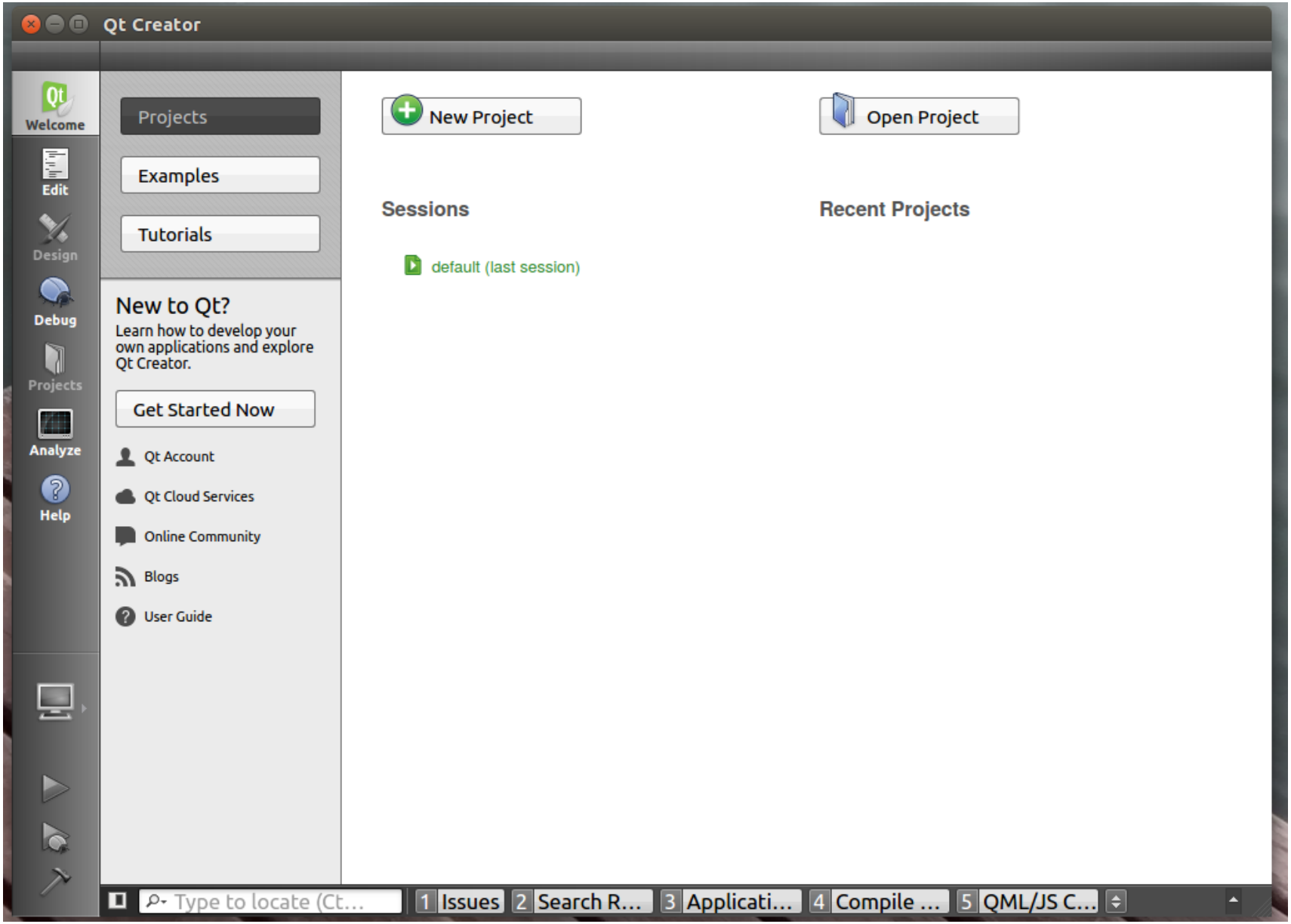
IDE's
Otra forma de manejar de forma automática el proceso de compilación es utilizando un *Intregrated Development Environment (IDE)*. Éstos permiten construir la estructura de un proyecto de forma amigable abstrayendonos de los detalles de compilación

IDE: QtCreator

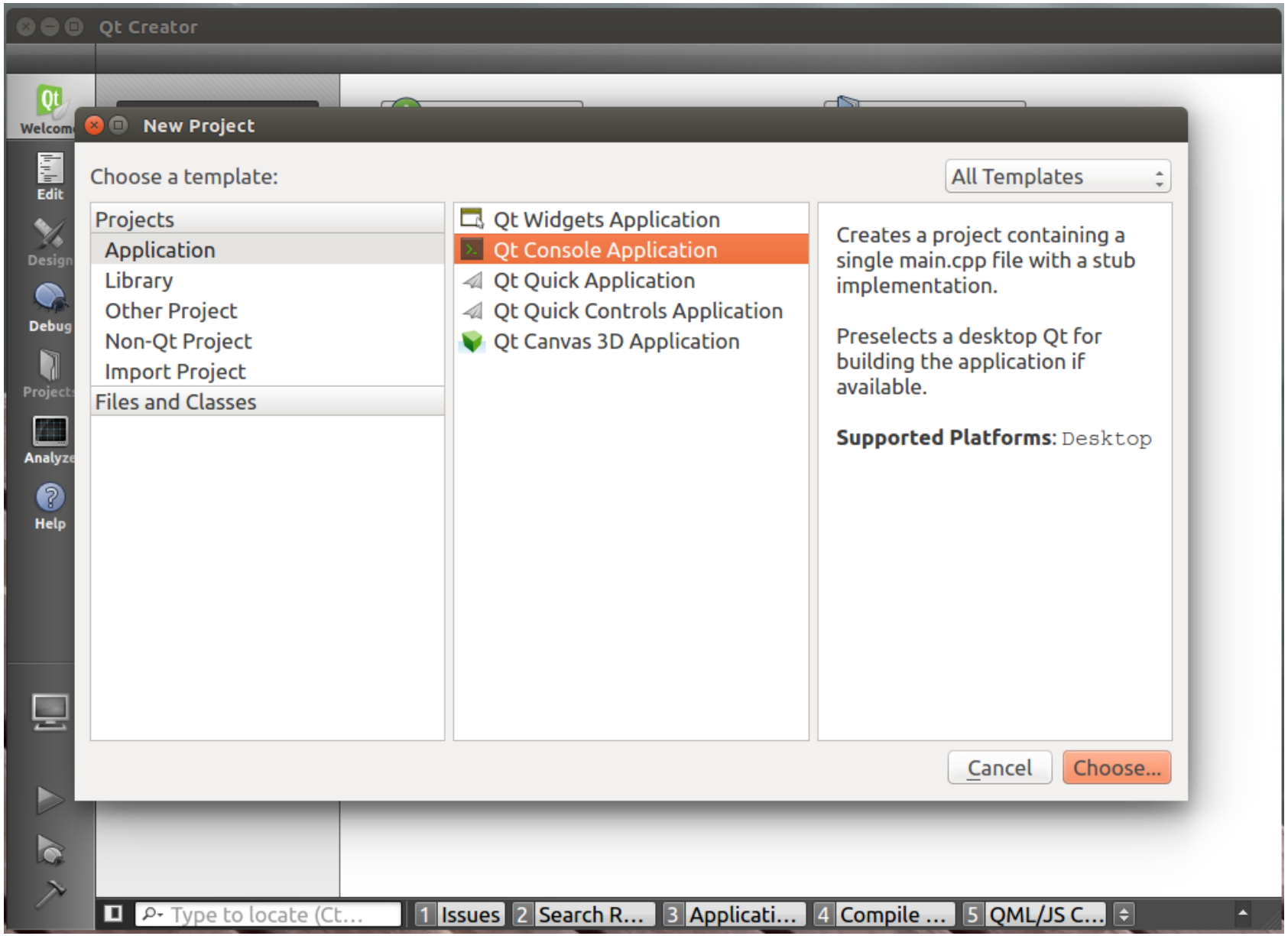
Instalación

- Desde la web: https://www.qt.io/download/
- En ubuntu: sudo apt-get install qt5-default qtcreator

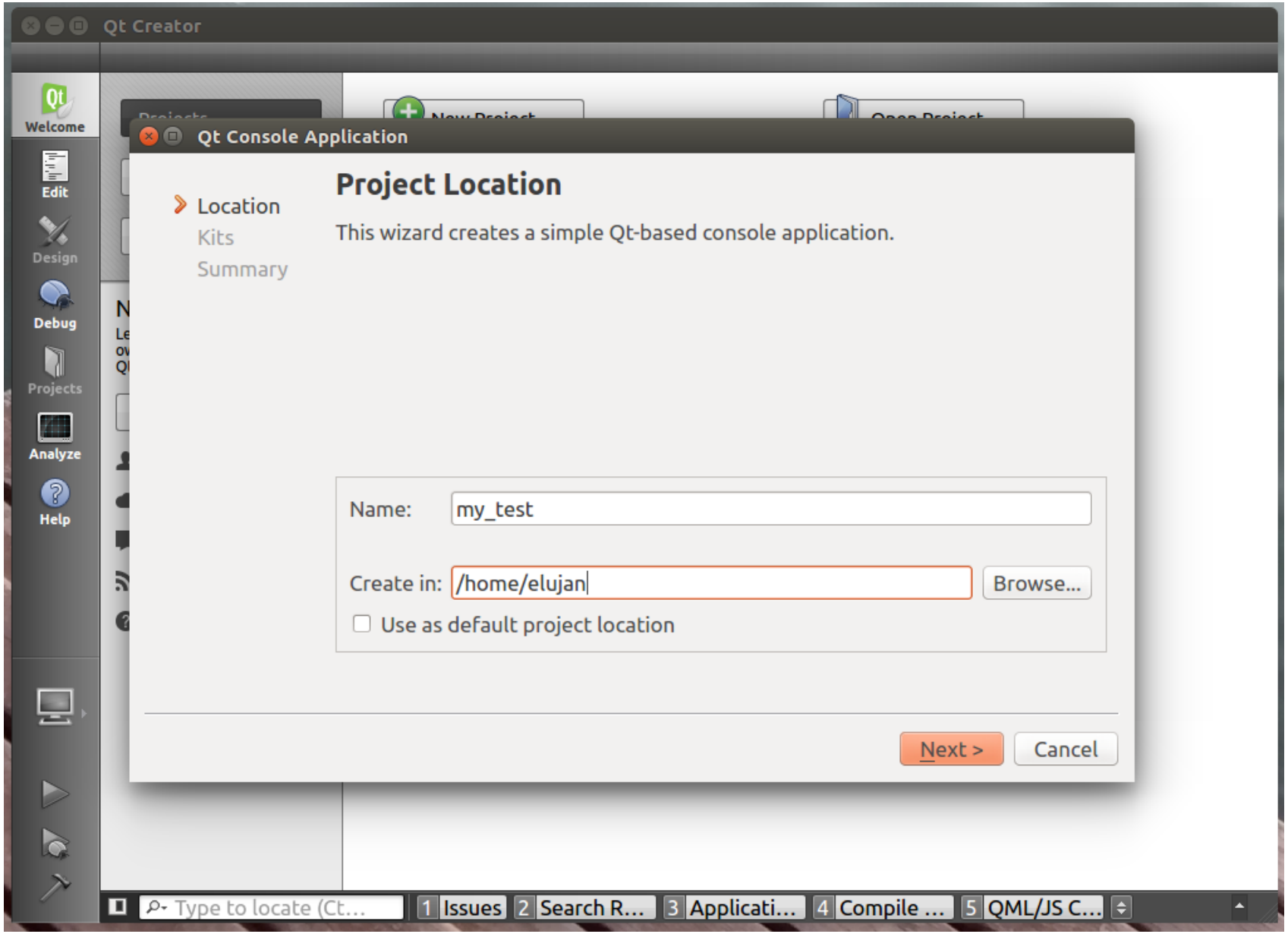
Ejemplo en Qt Creator



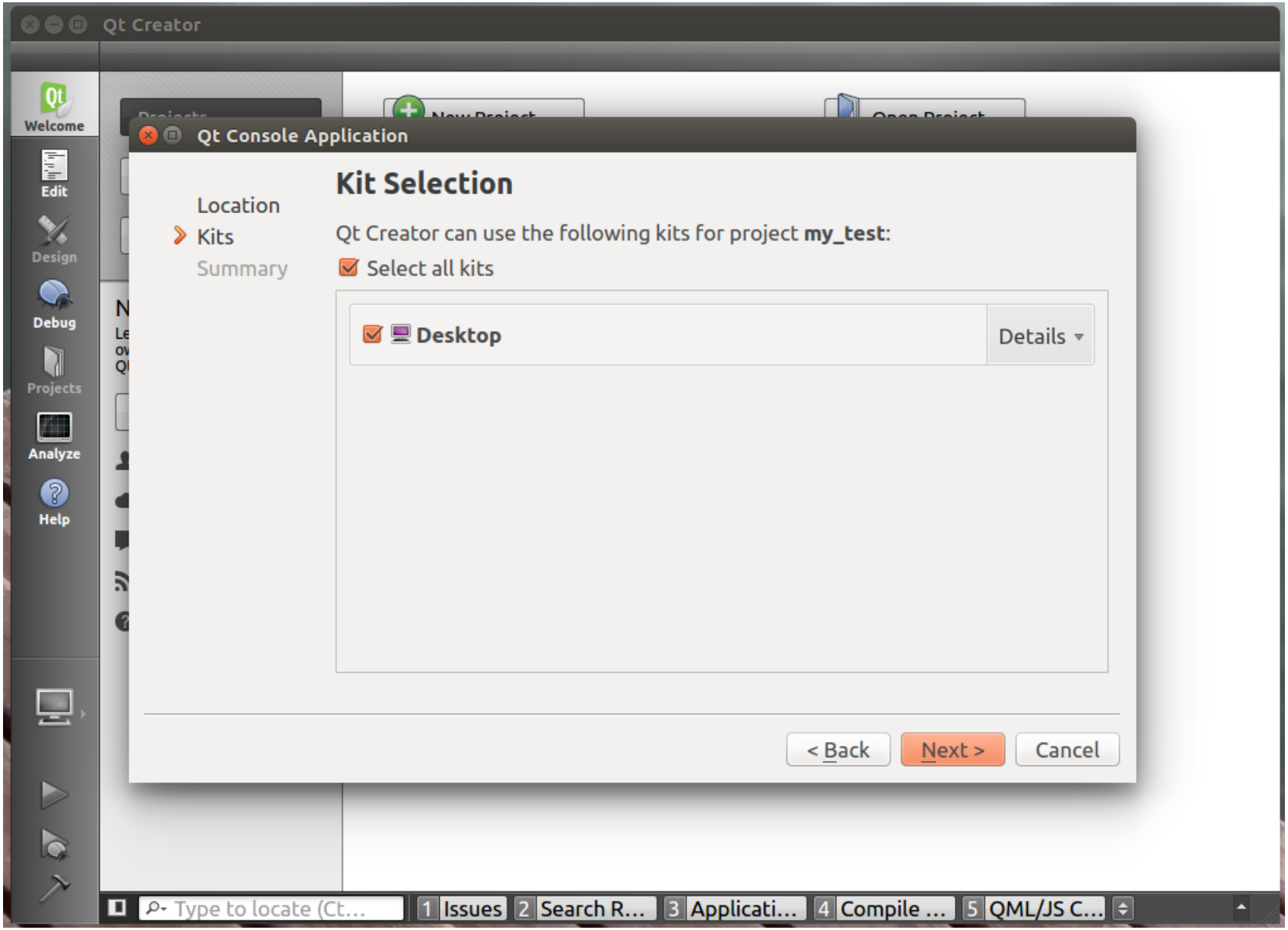
Ejemplo en Qt Creator



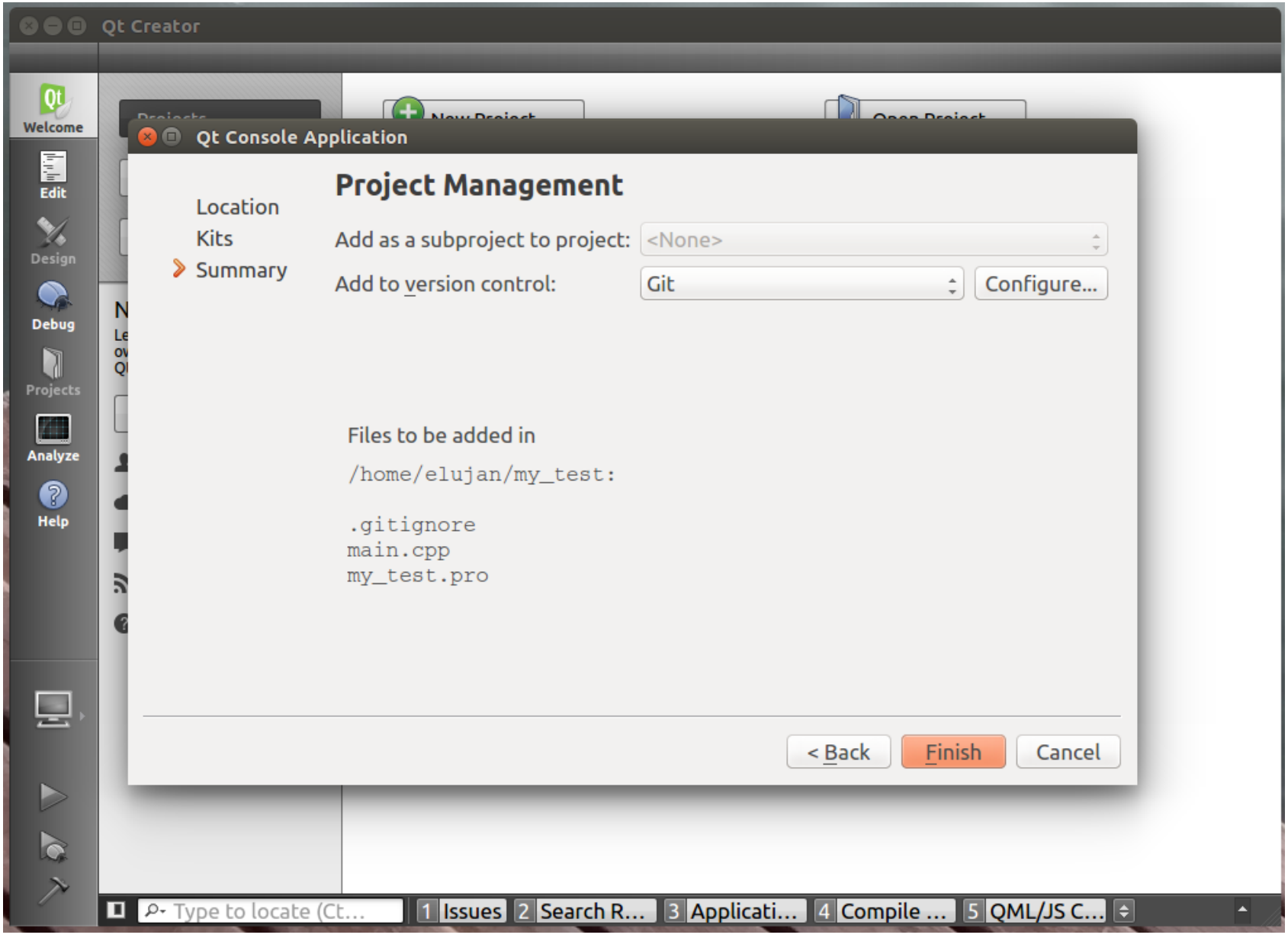
Ejemplo en Qt Creator



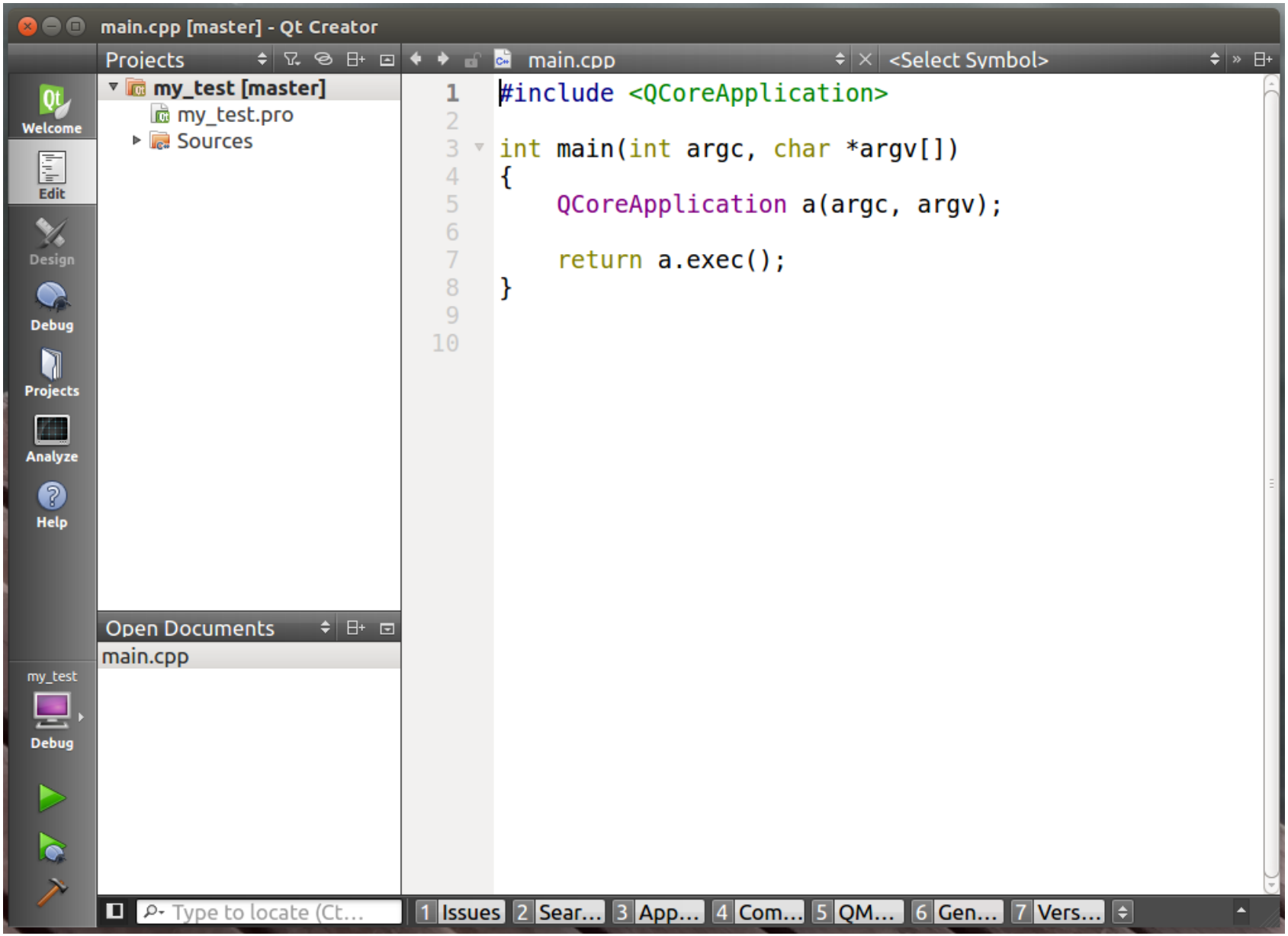
Ejemplo en Qt Creator



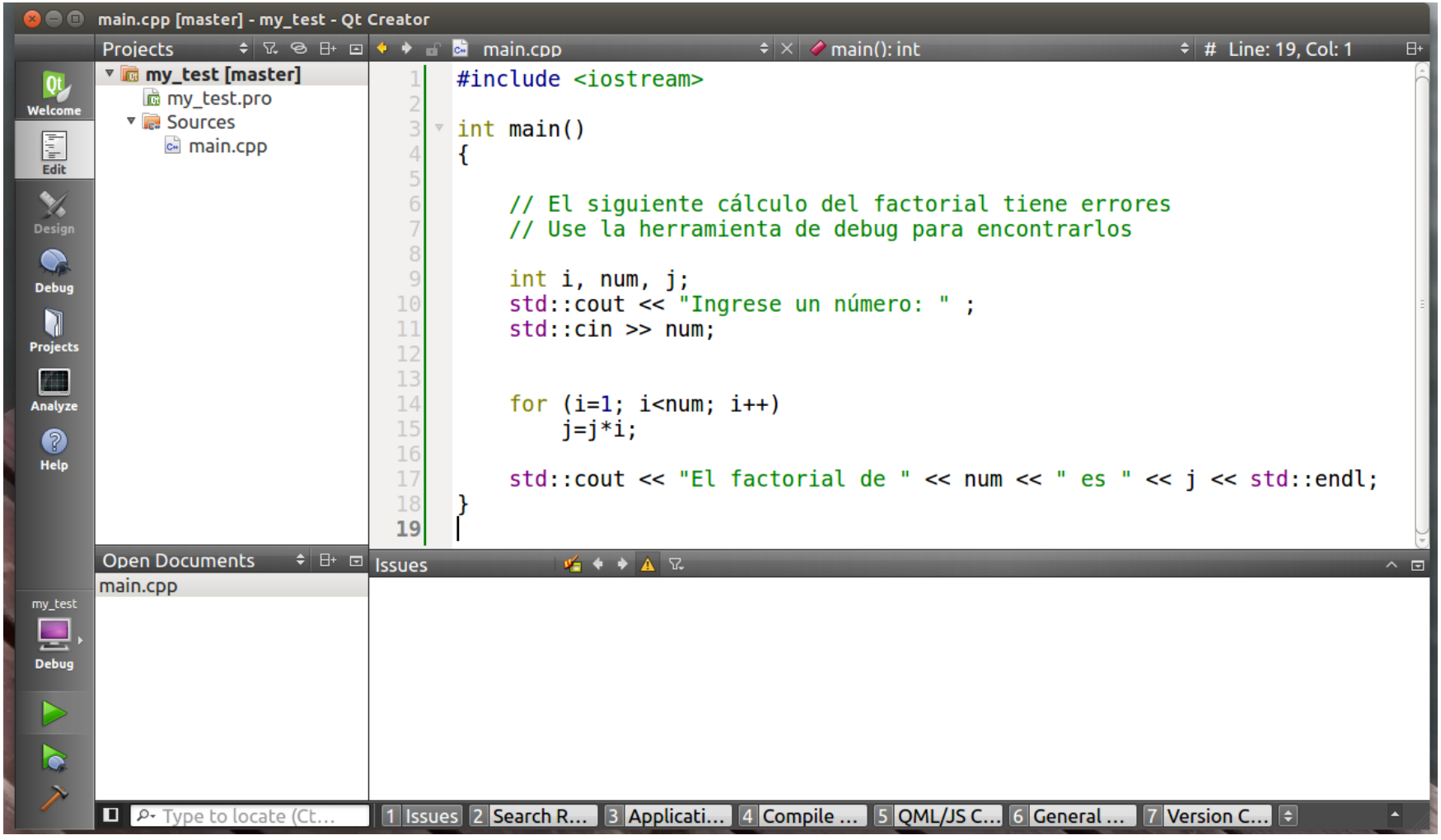
Ejemplo en Qt Creator



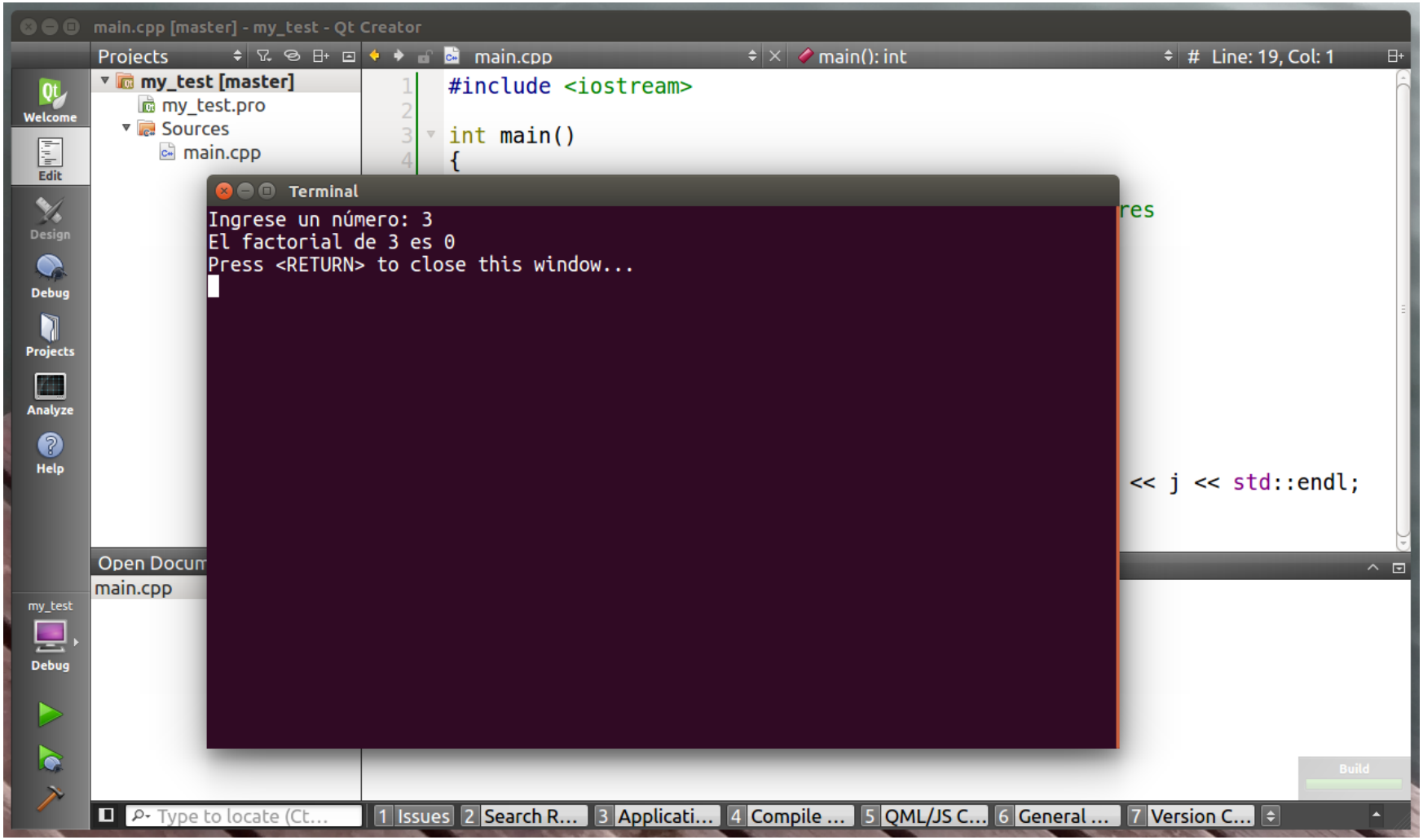
Ejemplo en Qt Creator



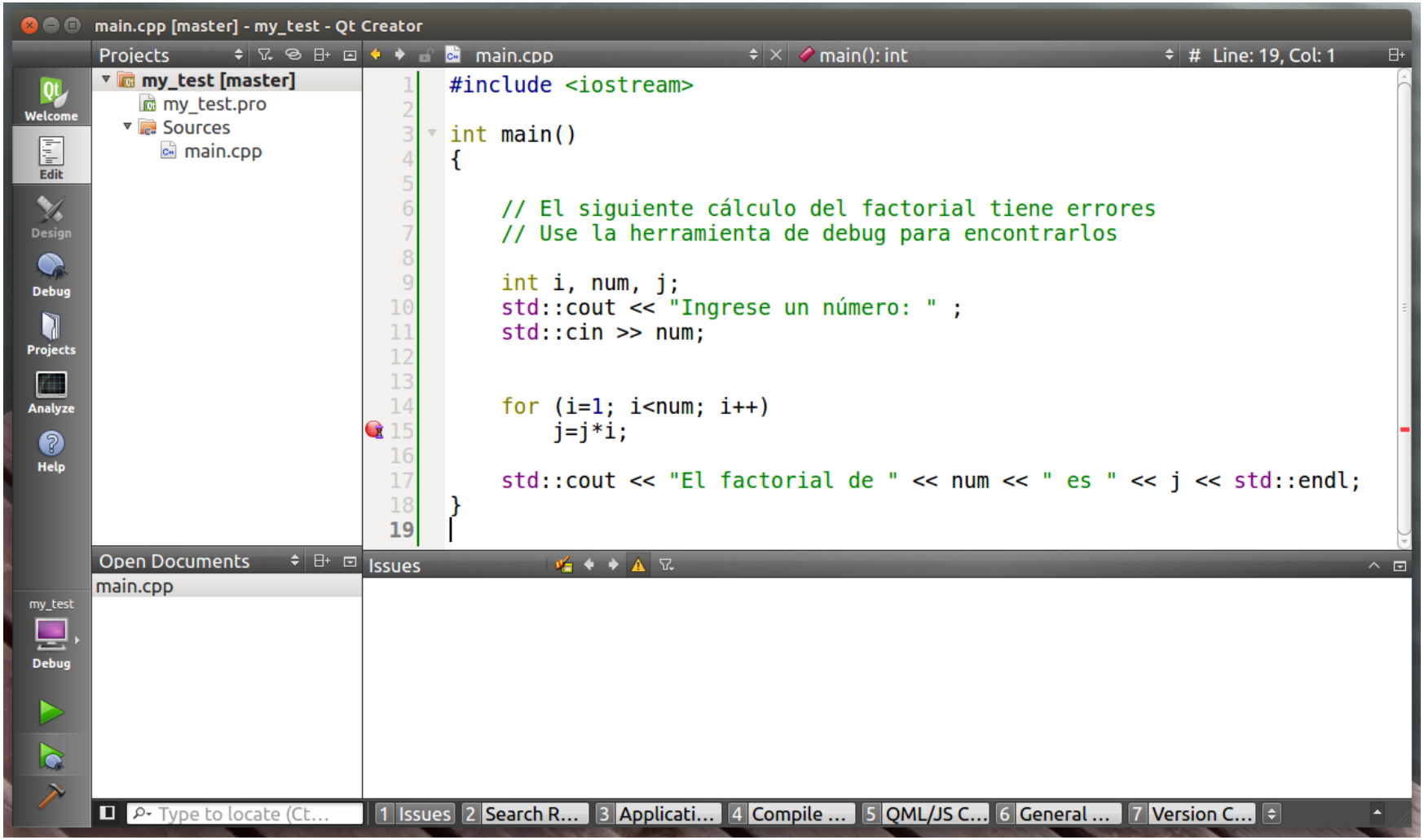
Ejemplo en Qt Creator



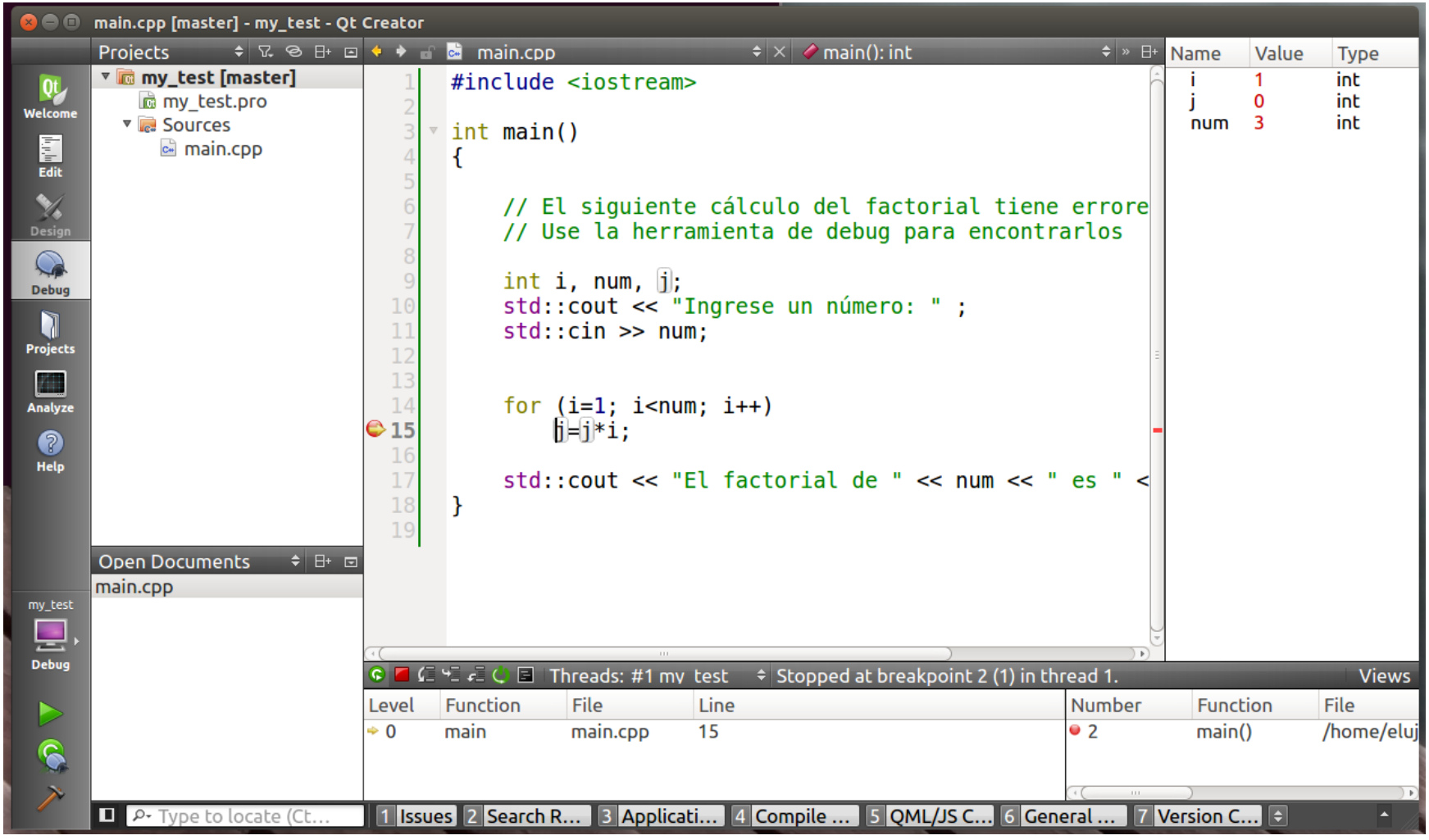
Ejemplo en Qt Creator



Ejemplo en Qt Creator



Ejemplo en Qt Creator



Test de unidad y Valgrind

Test de unidad:

- Implementar un test por función o método de mi clase
- Crear test independientes (mientras se pueda)
- Buscar casos borde donde pueda fallar
- Documentar el propósito de cada test y que chequea

Valgrind para chequear memory leaks:

- Compilar con información de debug:

```
$ g++ -g <archivos> -o binario
```

- Ejecutar por consola:

```
$ valgrind --leak-check=full -v ./binario
```