Reference Variables and Const Type Qualifier

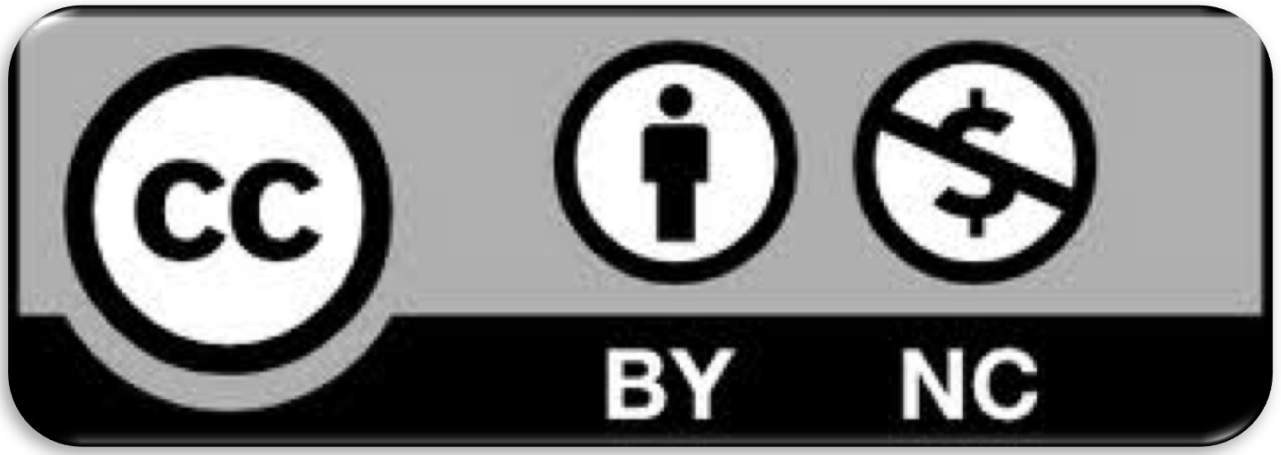
Alessandro Savino
Politecnico di Torino (Italy)

alessandro.savino@polito.it

www.testgroup.polito.it

License Information

This work is licensed under the Creative Commons BY-NC License



To view a copy of the license, visit:
http://creativecommons.org/licenses/by-nc/3.0/legalcode

Disclaimer

- **We disclaim any warranties or representations as to the accuracy or completeness of this material.**
- **Materials are provided "as is" without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.**
- **Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.**

Goal

– **This lecture presents a deeper view about C++ classes and objects**

Prerequisites

- **A basic knowledge about classes**

Homework

- **None**

Outline

- **Reference Variables**
- **Reference vs Pointers**
- **Const Type qualifier**

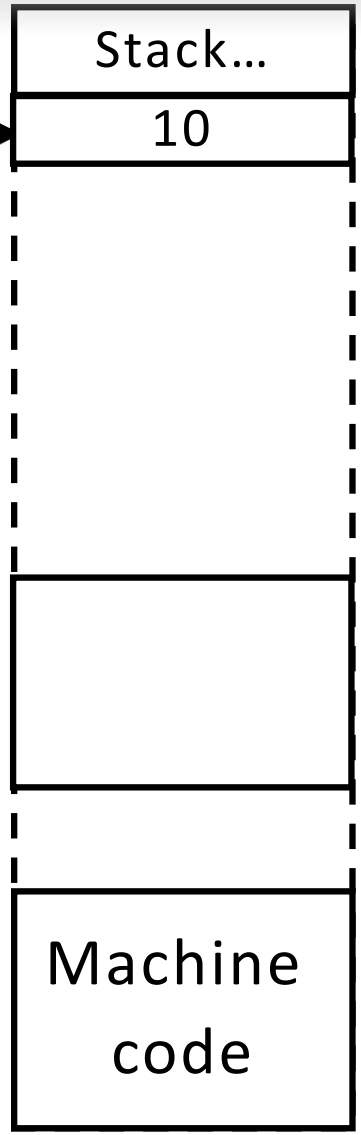
Parameters passed by reference

- **A reference variables is an alias for its corresponding argument in a function call.**
  - **`int &value`**

```
void f(int& r) {
  r=27;
};

int main() {
  int i=10;
  f(i);
  return i;
};
```

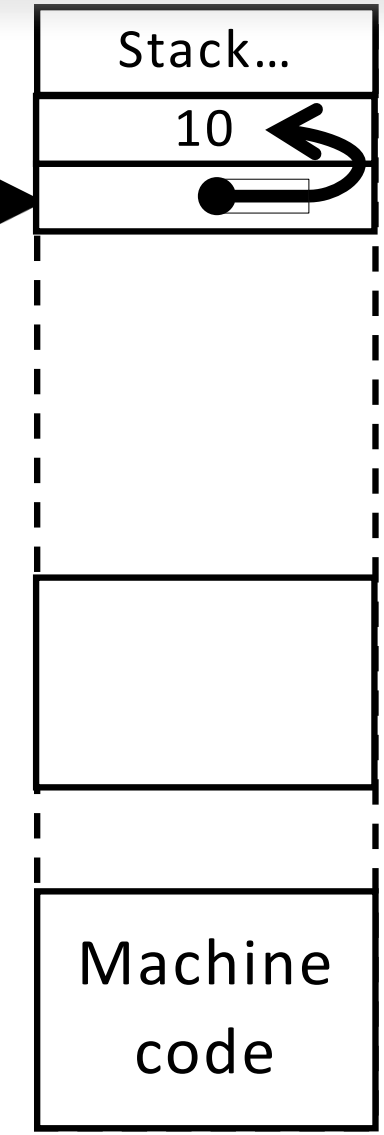
Parameters passed by reference

```
void f(int& r) {
   r=27;
};

int main() {
   int i=10;
   f(i);
   return i;
};
```



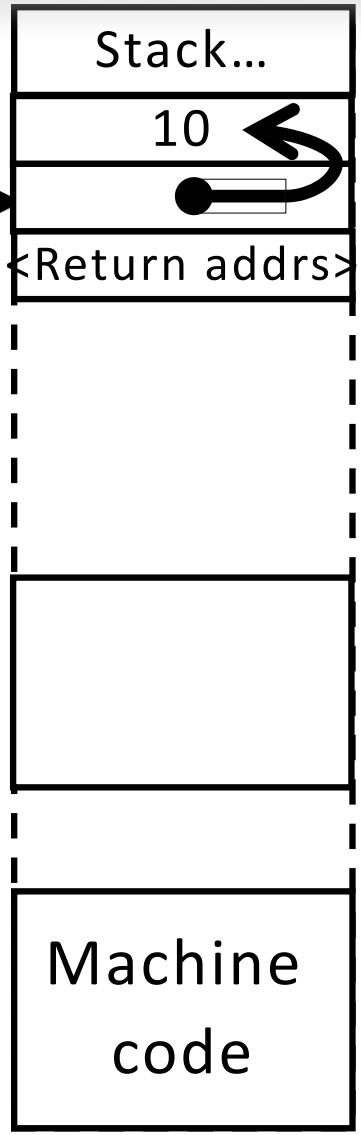
Parameters passed by reference

```
void f(int& r) {
   r=27;
};

int main() {
   int i=10;
   f(i);
   return i;
};
```



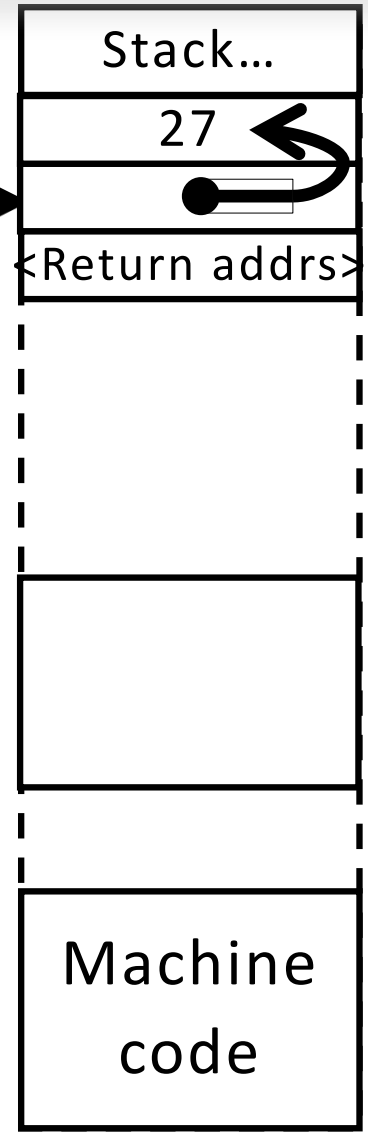
Parameters passed by reference

```
void f(int& r) {
   r=27;
};

int main() {
   int i=10;
   f(i);
   return i;
};
```



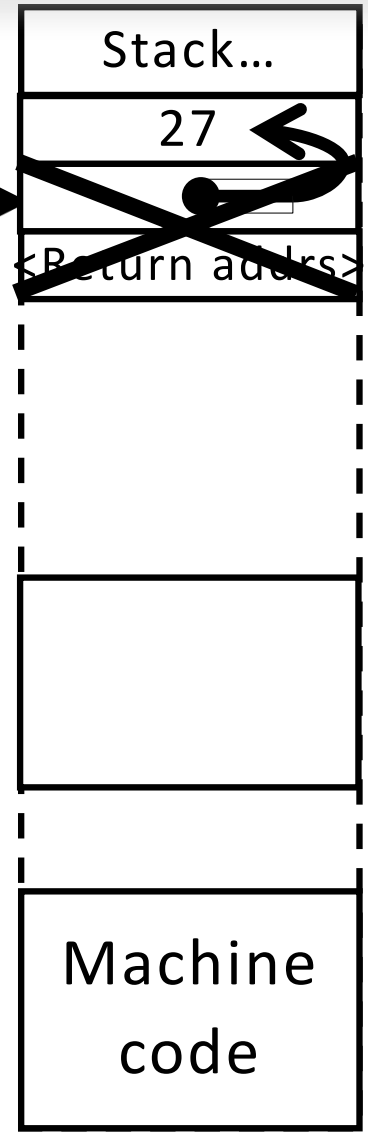
Parameters passed by reference

```
void f(int& r) {
   r=27;
};

int main() {
   int i=10;
   f(i);
   return i;
};
```



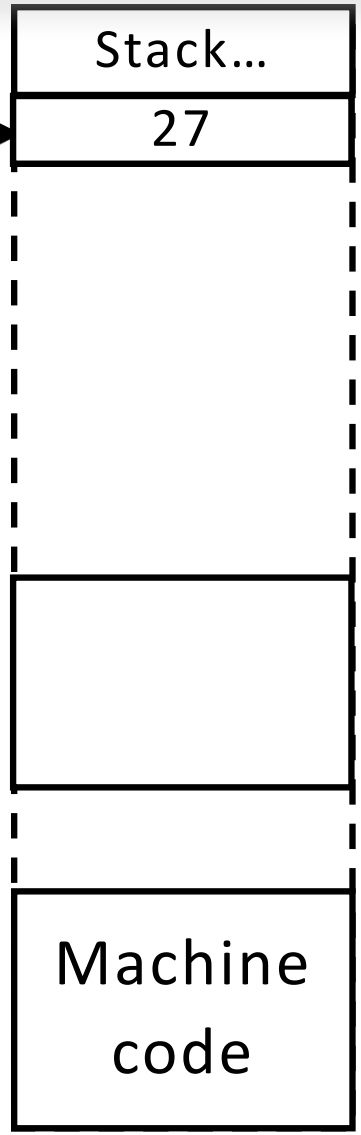
Parameters passed by reference

```
void f(int& r) {
   r=27;
};

int main() {
   int i=10;
   f(i);
   return i;
};
```



Parameters passed by reference

```
void f(int& r) {
   r=27;
};

int main() {
   int i=10;
   f(i);
   return i;
};
```



Outline

- Reference Variables
- Reference vs Pointers
- Const Type qualifier

Reference vs Pointers

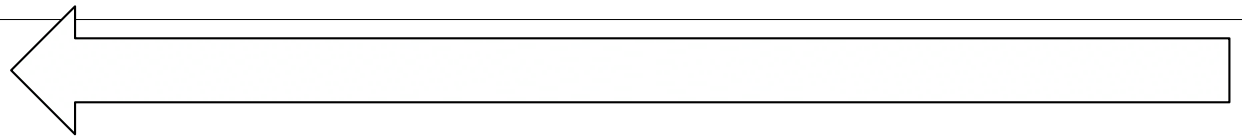
1. **A pointer can be re-assigned while a reference cannot, and must be assigned at initialization.**

```
int var;
int *p_to_var;
int &ref_to_var;
```

Reference vs Pointers

1. **A pointer can be re-assigned while a reference cannot, and must be assigned at initialization.**

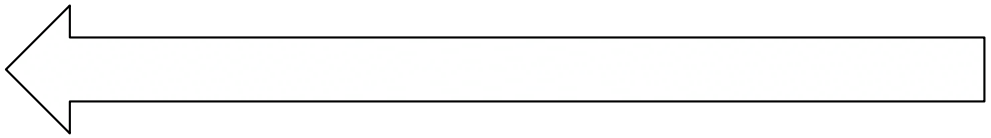
```
int var;
int *p_to_var;
int &ref_to_var;
```



Reference vs Pointers

1. **A pointer can be re-assigned while a reference cannot, and must be assigned at initialization.**

```
int var;
int *p_to_var;
int &ref_to_var;
```

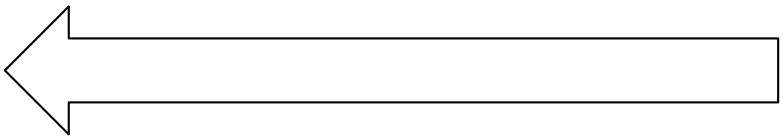


Valid

Reference vs Pointers

1. **A pointer can be re-assigned while a reference cannot, and must be assigned at initialization.**

```
int var;
int *p_to_var;
int &ref_to_var;
```

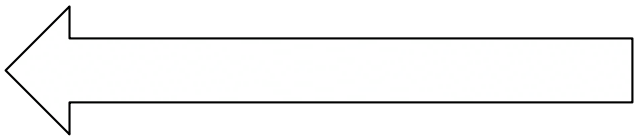


Not Valid

Reference vs Pointers

1. **A pointer can be re-assigned while a reference cannot, and must be assigned at initialization.**

```
int var;
int *p_to_var;
int &ref_to_var = var;
```



Valid

Reference vs Pointers

2. **You can have pointers to pointers to pointers offering extra levels of indirection. Whereas references only offer one level of indirection.**

```
int var;
int *p_to_var = &var;
int **p_to_p_to_var = &p_to_var;
```

Reference vs Pointers

3. **Pointer can be assigned nullptr directly, whereas reference cannot.**

```
int *p_to_var = nullptr;
int &ref_to_var = nullptr;
```

Reference vs Pointers

4. **Pointers can iterate over an array, you can use ++ to go to the next item that a pointer is pointing to, and + X to go to the X-th element. This is no matter what size the object is that the pointer points to.**
5. **A pointer needs to be dereferenced with * to access the memory location it points to, whereas a reference can be used directly. A pointer to a class/struct uses -> to access it's members whereas a reference uses a ..**

Reference vs Pointers

4. **Pointers can iterate over an array, you can use ++ to go to the next item that a pointer is pointing to, and + X to go to the X-th element. This is no matter what size the object is that the pointer points to.**
5. **A pointer needs to be dereferenced with * to access the memory location it points to, whereas a reference can be used directly. A pointer to a class/struct uses -> to access it's members whereas a reference uses a .**

Reference vs Pointers

- **Most important of all: pointers are memory location with their own address and space. References do not.**

```
int *p_to_var; // 4/8 bytes memory
int &ref_to_var; // just a label
```

Reference vs Pointers

- **Question: what is the advantage of all that?**

```
void swap(int *a, int
*b) {
  int temp = 0;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

```
void swap(int &a, int
&b) {
  int temp = 0;
  temp = a;
  a = b;
  b = temp;
}
```

Outline

- **Reference Variables**
- **Reference vs Pointers**
- **Const Type qualifier**

Const Type qualifier

- Using the keyword **const** to prevent **const** objects from getting mutated
- **Const is a very powerful keyword, allowing you to an advance control of your code.**
  - **Const member functions**
  - **Const parameters**
  - **Const references**
- **Extras: https://isocpp.org/wiki/faq/const-correctness**

Const member functions

- A **const member function** is a member function that guarantees it will not modify the object or call any non-const member functions (as they may modify the object).

```
class Rectangle {
public:
     …
    double getW() const;
    double getL() const;

    double getArea() const;
    double getPerimeter() const;
```

Const member functions

- We simply append the const keyword to the function prototype

```
class Rectangle {
public:
     …
    double getW() const;
    double getL() const;

    double getArea() const;
    double getPerimeter() const;
```

Const member functions

- We simply append the const keyword to the function prototype

Since they are just getter for internal data and calculation, it means they don't modify a Rectangle object.

```
class Rectangle
public:
     …
    double getW() const;
    double getL() const;

    double getArea() const;
    double getPerimeter() const;
```

Const member functions

- We need to repeat it in the implementation too

```
…
double Rectangle::getArea() const {
    return m_width * m_length;
}
…
```

Const member functions

- We need to repeat it in the implementation too

```
…
double Rectangle::getArea() const {
    return m_width * m_length;
}
…
```

Notice where the const keyword was written

Const member functions

- We need to repeat it in the implementation too

```
…
double Rectangle::getArea() const {
    return m_width * m_length;
}
…
```

Notice the operation does not change the object

Const parameters

- You can set as cost any parameter of member function
  - It means you expect not to modify it

```
class Rectangle {
public:
     Rectangle();
     Rectangle(const double w, const
double l);
...
```

Const parameters

- You can set as cost any parameter of member function
  - It means you expect not to modify it

```
class Rectangle {
public:
     Rectangle();
     Rectangle(const double w, const
double l);
...
```

Does it make any sense?

Const parameters

- You can set as cost any parameter of member function
  - It means you expect not to modify it

```
class Rectangle {
public:
     Rectangle();
     Rectangle(const double w, const
double l);
...
```

Actually, since they are passed by value, it only serves as reminder that they won't be touched...

Const references

- Const references are meant to replace passing the parameter by value to avoid the copy of it
  - Still you must guarantee that no further modification will be done!

```
class Rectangle {
public:
     ...
    void setW(const double &w);
    void setL(const double &l);
```

Const references

- You must keep the const in the implementation...

```
void Rectangle::setW(const double &w) {
    m_width = w;
}

void Rectangle::setL(const double &l) {
    m_length = l;
}
```

Const references

- You must keep the const in the implementation...

Be aware that in the implementation of the function you can resort only to const methods of the object!

```
void Rectang    .:setW(const double &w) {
    m_width = w;
}

void Rectangle::setL(const double &l) {
    m_length = l;
}
```


Малые Автюхи, Калинковичский район, Республики Беларусь